



**BRNO UNIVERSITY OF TECHNOLOGY**

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA**

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

**CODEC DETECTION FROM SPEECH**

DETEKCE KODEKU Z ŘEČOVÉHO SIGNÁLU

**BACHELOR'S THESIS**

BAKALÁŘSKÁ PRÁCE

**AUTHOR**

AUTOR PRÁCE

**JOSEF JON**

**SUPERVISOR**

VEDOUCÍ PRÁCE

**Doc. Dr. Ing. JAN ČERNOCKÝ**

**BRNO 2017**

**Brno University of Technology - Faculty of Information Technology**

Department of Computer Graphics and Multimedia

Academic year 2016/2017

**Bachelor's Thesis Specification**

For: **Jon Josef**

Branch of study: Information Technology

Title: **Codec Detection from Speech**

Category: Speech and Natural Language Processing

**Instructions for project work:**

1. Get acquainted with the principles of speech coding.
2. Obtain a set of normalized codecs (from ITU, ETSI, 3GPP, etc.).
3. Simulate the influence of codecs on a set of speech recordings.
4. Get acquainted with classification by artificial neural networks, select a suitable toolkit and train a classifier of codecs.
5. Test on simple data (simulation with codecs) and more complicated cases (sequence of codecs, real signals).
6. Analyze the results, suggest and implement improvements.
7. Create a poster and/or short video presenting your work.

**Basic references:**

- ZRE lectures <http://www.fit.vutbr.cz/study/courses/ZRE/>
- A. Spanias: Speech Coding: A Tutorial Review, Proc. of the IEEE, Vol 82, No 10, October 1994.
- based on supervisor's recommendation.

**Requirements for the first semester:**

Items 1, 2, 3, partial fulfilment of item 4.

Detailed formal specifications can be found at <http://www.fit.vutbr.cz/info/szz/>

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

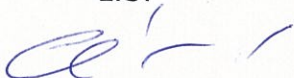
Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Černocký Jan, doc. Dr. Ing.**, DCGM FIT BUT

Beginning of work: November 1, 2016

Date of delivery: May 17, 2017

**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**  
Fakulta informačních technologií  
Ústav počítačové grafiky a multimédií  
602 00 Brno, Božetěchova 2  
L.S.



---

Jan Černocký  
Associate Professor and Head of Department

## Abstract

This thesis deals with codec detection from compressed speech signal. The primary goal was to identify which features distinguish selected codecs, and then create an environment facilitating experiments with various types of classifiers and their configurations. Support vector machines and neural networks, modeled using the Keras library, were used. The main contribution of this work is the experimental part, in which the effects of the neural networks parameters are discussed. After tuning the parameters and finding their optimal values, the network achieved accuracy over 98% on a test set comprising data from six different codecs.

## Abstrakt

Tato práce se zabývá detekcí kodeků z komprimovaného řečového signálu. Cílem bylo zjistit, jaké charakteristiky rozlišují jednotlivé kodeky a následně vytvořit prostředí vhodné pro experimenty s různými typy a konfiguracemi klasifikátorů. Použity byly Support vector machines a především neuronové sítě, které byly vytvořeny pomocí nástroje Keras. Hlavním přínosem této práce je experimentální část, ve které je analyzován vliv různých parametrů neuronové sítě. Po nalezení nejvhodnější kombinace parametrů dosáhla síť přesnosti klasifikace přes 98% na testovací sadě obsahující data z 6 kodeků.

## Keywords

Neural networks, codec classification, speech processing, LPC, Keras, machine learning, Support vector machines, SVM, GRU, LSTM, codec

## Klíčová slova

Neuronové sítě, klasifikace kodeků, zpracování řeči, LPC, Keras, strojové učení, Support vector machines, SVM, GRU, LSTM, kodek

## Reference

JON, Josef. *Codec Detection from Speech*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. Dr. Ing. Jan Černocký

# Codec Detection from Speech

## Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Mr. Jan Černocký. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....

Josef Jon  
May 18, 2017

## Acknowledgements

I wish to express my thanks to my supervisor, Jan Černocký, for his guidance and encouragement.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	State of the art . . . . .	3
1.2	Content overview . . . . .	4
<b>2</b>	<b>Theory</b>	<b>5</b>
2.1	Codecs . . . . .	5
2.1.1	Vocal tract . . . . .	6
2.1.2	Waveform codecs . . . . .	6
2.1.3	Parametric codecs . . . . .	7
2.1.4	Hybrid codecs . . . . .	8
2.2	Codecs used in the experiments . . . . .	8
2.3	Used features . . . . .	10
2.4	Classification . . . . .	11
2.4.1	SVM . . . . .	11
2.4.2	Neural Networks . . . . .	12
2.4.3	Feedforward neural networks . . . . .	14
2.4.4	Recurrent neural networks . . . . .	15
2.4.5	Metrics . . . . .	20
<b>3</b>	<b>Data</b>	<b>22</b>
<b>4</b>	<b>Implementation</b>	<b>24</b>
4.1	Used tools . . . . .	24
4.1.1	Soundfile . . . . .	24
4.1.2	Yaafe . . . . .	24
4.1.3	SciPy . . . . .	25
4.1.4	Keras . . . . .	25
4.2	Data preprocessing . . . . .	27
4.3	Feature extraction . . . . .	27
4.3.1	Feature statistical description . . . . .	28
4.4	Classifiers . . . . .	28
4.4.1	SVM . . . . .	28
4.4.2	Neural networks . . . . .	29
<b>5</b>	<b>Experiments</b>	<b>30</b>
5.1	Hardware . . . . .	30
5.2	Network architectures . . . . .	30
5.3	Data . . . . .	31

5.4	Features . . . . .	31
5.5	Baseline results . . . . .	31
5.6	Improving the networks . . . . .	34
5.6.1	Possible improvements . . . . .	34
5.6.2	Hyperparameters . . . . .	34
5.6.3	Data manipulation . . . . .	37
5.6.4	Different architecture . . . . .	39
5.6.5	Optimal networks . . . . .	39
5.7	Further experiments . . . . .	40
5.7.1	Features . . . . .	40
5.7.2	„Production“model . . . . .	42
5.7.3	Shorter window . . . . .	42
5.8	More subsequent encodings . . . . .	42
<b>6</b>	<b>Conclusions</b>	<b>44</b>
6.1	Summary . . . . .	44
6.2	Future work . . . . .	44
6.2.1	Easily achievable goals . . . . .	44
6.2.2	Long term goals . . . . .	45
	<b>Bibliography</b>	<b>46</b>
<b>A</b>	<b>Cookbook</b>	<b>49</b>
A.1	Software and modules . . . . .	49
A.1.1	Python . . . . .	49
A.1.2	Other tools . . . . .	50
A.1.3	Scripts . . . . .	50
A.1.4	Step by step . . . . .	51

# Chapter 1

## Introduction

In today's world, spoken communication is predominantly transmitted in a digitized form. The need for good quality of speech at reasonable bandwidth usage led to development and standardization of numerous algorithms, which are used to minimize a signal bit rate without compromising its quality – codecs. The goal of this work is to present a way of identifying a codec used to encode a speech signal using only the parameters of a decoded signal, without the knowledge of the original signal.

Identifying the codec in case we do not have access to a file header can have many possible applications. Since codec selection has a big influence on the perceptual quality of speech signal, it also has a great impact on the quality of automated speech recognition (ASR), as proven for example by Besacier et al. [8]. If a codec in use could be detected, ASR would be able to use models designed specifically for this codec and thus improve its accuracy. Another field that could benefit from this work is IP telephony. Knowledge of the codec can be used to refine the accuracy of estimation of bandwidth needed by a VoIP connection – different codecs need different bit rates to achieve acceptable speech quality. Another possible application is in law enforcement. Since many different codecs are commonly used today, we can identify the source of the signal if we know which codecs were used or detect if the signal was tampered with on its way.

This work will present a technique based on speech features and neural networks classification algorithms. Input signals are classified by two distinct neural network architectures, (both using a different approach to feature extraction). In the first step, a speech corpus was encoded with six commonly used codecs – G.711, G.723.1, G.729, GSM-EFR, Speex and MP3. Then the signal is framed and various speech features are extracted using freely available software. These features were processed in two different ways. The first approach was to construct a feedforward neural network operating on the statistical description of the features – skewness, kurtosis, mean and variance. The second approach was to design a recurrent neural network and feed the input frames as a sequence of feature vectors into it. The input files were classified by the neural networks (first separately, then by a joint network utilizing both approaches) and the results were compared with each other as well as with results presented in another research papers.

### 1.1 State of the art

Despite the importance of this topic, there is only a limited amount of research papers available regarding codec identification. A work by Sharma et al. [27] shows a method

based on feedforward neural network which uses statistical description of various features as their input, achieving 92% accuracy in classification of 5 codecs and their bitrates, even with added noise. The feedforward network part of my thesis was inspired by this paper.

A team formed by Samet Hicsonmez, Husrev Sencar and Ismail Avcibas [16] used a completely different approach – they use mathematical description of the randomness and chaotic features in the signal, reaching 95% classification accuracy on 16 commonly used codecs.

Another approach was presented by Scholz, Leutelt and Heute [26]. In their paper, they decompose the signal into harmonic and noise parts and detect distortions typical for specific codecs in each part. Their classifier achieved around 92% accuracy on 5 codecs.

## 1.2 Content overview

This first introductory chapter provides overview of the state of the art and the structure of this thesis.

The following chapter deals with the theory needed to create a functional experimental environment. Codecs used in the experiments are outlined, together with principles of encoding needed to understand the features that will be used as the inputs for the classifiers. The second part of the chapter contains an overview of the classifiers that will be implemented, especially the description of the two neural network architectures. Metrics employed to evaluate the models are described in the final part of this chapter.

The third chapter contains description of training and test data. Origin, parameters and sizes of datasets can be found in this chapter, alongside with description of a method used to encode them with different encoders.

The fourth chapter describes the implementation of classifiers discussed in Chapter 2. It is an overview of what tools were used to construct the experimental environment. The first part deals with feature extraction, how the features from Chapter 2 were obtained and what preprocessing had to be done. The second part introduces Keras, a library used to build the neural networks.

When the environment is ready, we can run the experiments. That is what Chapter 5 is about - all of the experiments are described and their results are presented there. The final chapter 6 summarizes the experiments and provides some possibilities of future development of this work.



# Chapter 2

## Theory

This chapter serves as an overview of the principles I had to study in order to be able to carry out the experiments. In the first part, techniques of speech encoding are explained, and all of the codecs that will be used later on, are briefly commented. In the second part, I look at the problem from machine learning perspective and suitable classification methods are discussed. The complete pipeline for classifying the files is the following:

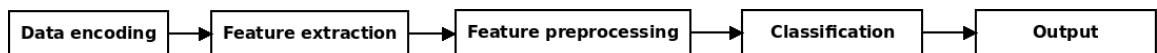


Figure 2.1: Classification pipeline

### 2.1 Codecs

The goal of this work is the identification of a codec used to encode a signal. The aim of speech compression is to minimize the bitrate of a signal while keeping the speech quality at desired level. There are two main approaches to speech compression: the waveform codecs, which aim to exploit the redundancies in speech samples when coding and reconstruct the original waveform as closely as possible during the decoding, and parametric codecs, which use the knowledge of how speech is produced in human vocal tract to create a model and represent the speech as a set of this models parameters. In table 2.1, some of the terms used in following text are defined.

Name	Bitrate
Very low bitrate	< 2.4 kb/s
Low bitrate	2.4 - 8 kb/s
Medium bitrate	8-16 kb/s
High bitrate	> 16 kb/s

Table 2.1: Bitrate ranges used later <sup>1</sup>

---

<sup>1</sup> Bitrate classification taken from slides at [http://www.fit.vutbr.cz/study/courses/ZRE/public/pred/06\\_kod/06\\_kod.pdf](http://www.fit.vutbr.cz/study/courses/ZRE/public/pred/06_kod/06_kod.pdf)

### 2.1.1 Vocal tract

Since the examined codecs exploit the nature of how human speech is generated, it is useful to have at least basic understanding of the human vocal tract. The sound is produced when air is pushed from lungs through the vocal cords and mouth. There are two types of sounds in human speech – voiced and unvoiced. For voiced sounds, the local cords vibrate at a given frequency, called the pitch frequency, whereas for unvoiced sounds, the vocal cords stay open. A simplified human vocal tract scheme can be seen in Figure 2.2.

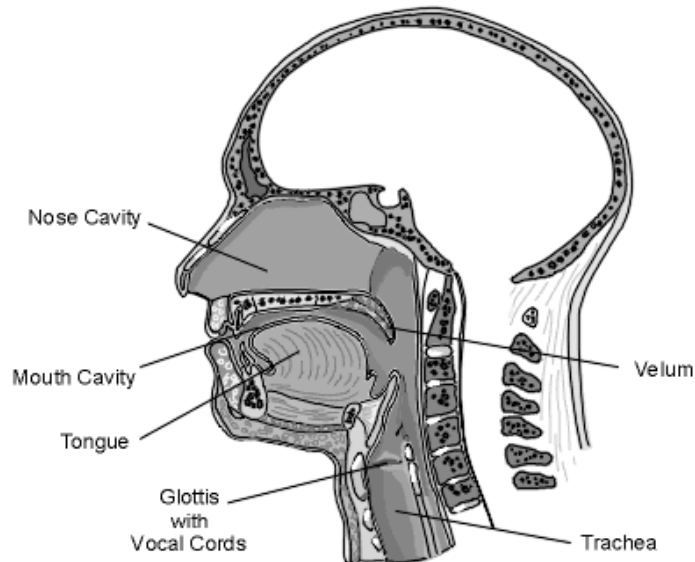


Figure 2.2: A scheme of a human vocal tract.<sup>2</sup>

### 2.1.2 Waveform codecs

Waveform codecs work by removing the redundancies and correlation in signals waveform between speech samples. The most typical example is PCM (Pulse code modulation), standardized by ITU-T G.711 standard [1], which uses logarithmic compression to quantize 14 or 13 bit samples down to 8 bits. The scale is finer at lower signals since it is known that smaller signals are more represented in a speech.. This codec will be described in more detail later. Another waveform codec is ADPCM (Adaptive differential pulse code modulation) which in addition to ordinary PCM also predicts the signal from previous samples and encodes only the error of the prediction. This approach removes correlation between adjacent parts of signal, thus reaching further compression. The block diagrams of ADPCM encoder and decoder are shown in Figure 2.3. The waveform codecs generally operate at higher bitrates than parametric codecs, typically around 32 kb/s, and the speech loses intelligibility at bitrates below 16 kb/s.

---

<sup>2</sup>[https://www2.spsc.tugraz.at/add\\_material/courses/sc1/vocoder/](https://www2.spsc.tugraz.at/add_material/courses/sc1/vocoder/)

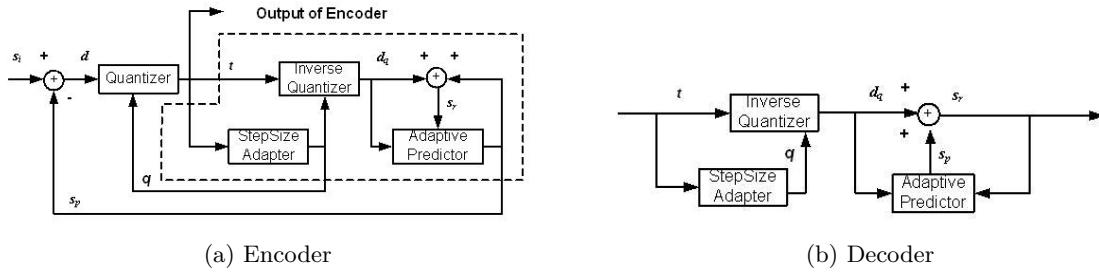


Figure 2.3: ADPCM encoder and decoder block diagrams<sup>3</sup>

### 2.1.3 Parametric codecs

To compress speech further, parametric codecs use models of the human speech production to encode the signal. Instead of signal waveforms, they only send parameters of these models, which are used on the receiving side to reconstruct the speech. These parameters occupy significantly less space than compressed waveforms, which allows them to use much lower bit rates, starting at 800 b/s. The typical representative of these codecs is a technique call Linear prediction coding – LPC.

#### LPC

Linear predictive coding is one of the most widely used methods for speech analysis in low or medium bitrate parametric coders [30]. It exploits the nature of speech production in human vocal tract, modeling it using LPC filter. The main premise of LPC analysis is that the human vocal tract can be represented as a buzzer (glottis) at the end of the tube (throat and mouth). Its goal is to remove redundancies in a speech signal, thus lowering the output bitrate. Let's see how the human speech can be approximated using a simple mathematical model, as demonstrated in Figure 2.4. In the first step, the signal is framed, usually by 20 ms. The next step is to generate the excitation signal, which will later be modified by a filter. For each frame, the model needs to decide whether the contained speech is voiced (i.e. vocal cords vibrate when producing the sound) or unvoiced (vocal cords are not involved). In case the frame is voiced, the pitch period (frequency of the vocal cords vibration) is estimated and a periodic pulse train at this frequency is used as an excitation signal (mimicking the vocal cords). Otherwise the model generates a white noise. Then, this excitation signal is multiplied by signal energy, also referred to as gain, and filter parameters are estimated by the LPC method relying on assumption that each sample can be estimated as a linear combination of the previous samples. The LPC filter has a form of  $1/A(z)$  filter and mainly reflects the spectral envelope of the speech signal. Its coefficients represent formants of the signal. After the estimation, following information is sent to the decoding side – pitch period (7 bits in LPC-10), voicing decision bit, gain (5 bits in LPC-10) and vocal tract model coefficients (41 bits in LPC-10). In the decoder, these parameters are used to synthesize an output speech signal.

<sup>3</sup><http://electrotech99.blogspot.cz/2011/01/adaptive-differential-pulse-code.html>

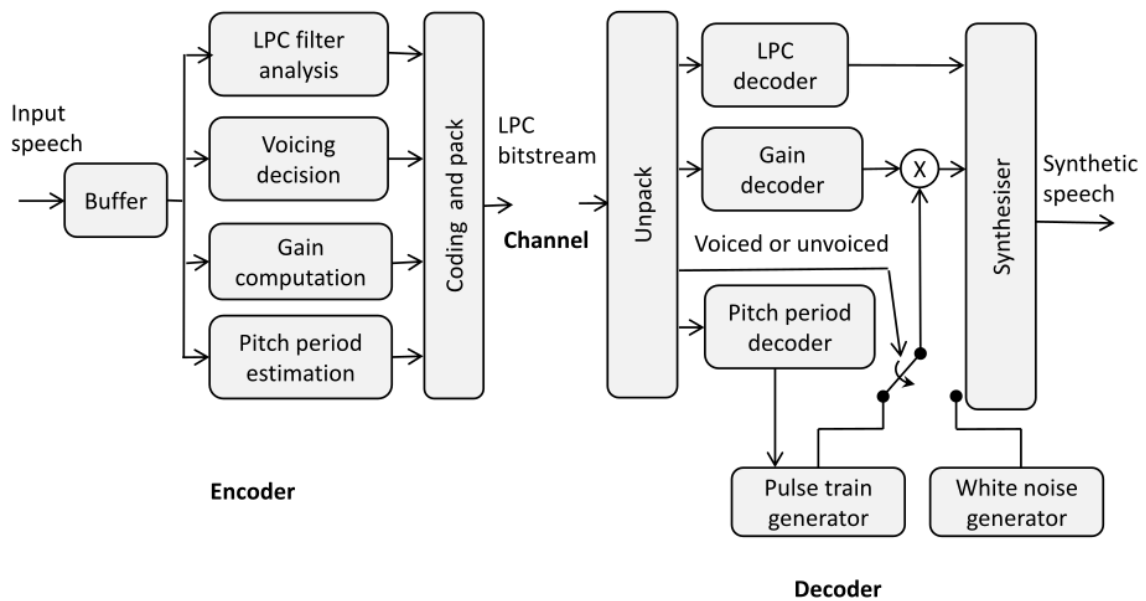


Figure 2.4: LPC encoding and decoding<sup>4</sup>

#### 2.1.4 Hybrid codecs

Parametric codecs can operate on very low bitrates while keeping intelligibility. However, the reconstructed speech sounds somewhat mechanical, primarily due to binary voiced/unvoiced decision, and the fact that the excitation signal is not a simple periodic pulse wave. These issues are addressed by hybrid encoders, which use features from both waveform and parametric codecs to achieve better perceptual quality. The way to manage that is to match the excitation signal of the original speech as closely as possible. The approach used in hybrid codecs is called Analysis-by-Synthesis – the codec features a speech synthesizer (LPC and pitch synthesis) and performs a search in a database of waveforms, which is called a codebook. Index of the best match is then sent together with other parameters to the decoding side.

#### CELP

One of the most widely used techniques in hybrid compression coding is Code-Excitation Linear Prediction (CELP)[30]. CELP coders contain codebooks of 256 – 1024 samples at both encoding and decoding ends. It is often split into two codebooks, one containing waveforms and one containing multi-pulse excitation signals.

## 2.2 Codecs used in the experiments

After previous brief summary of the techniques used in speech compression, this section shows, how are these techniques used in practice. The codecs that will be used later on in the experiments are presented in the next few paragraphs.

<sup>4</sup>Figure from Guide to Voice and Video over IP book [30]

**G.711** G.711 [1] is an ITU-T standard describing the PCM codec, used primarily in telephony. It is a waveform codec with narrowband sampling rate (8 kHz) and sample size of 8 bits, which results in bitrate of 64 kb/s. PCM provides toll-quality speech and it is often used as “baseline” for evaluating low-bitrate parametric codecs, both regarding speech quality and compression ratio. Two different compressions are used – mu-law, which is used primarily in North America and Japan, and A-law, which is used in rest of the world. The PCM encoder converts 14 bit (mu-law) or 13 bit (A-law) linear PCM samples to 8 bit samples using logarithmic quantization, which allows finer scale on low-level speech and coarser scale on higher frequencies, thus using the available sample size more effectively since most of the human speech takes part in the lower parts of used frequencies.

**G.723.1** ITU-T standard G.723.1 [4] was designed mainly for use in low bitrate telephony applications and offers two algorithms operating on different bitrates for better flexibility. Algebraic CELP (ACELP) for 5.3 kb/s bit rate and Multi Pulse-Maximum Likelihood Quantisation (MP-MLQ) for 6.3 kb/s bit rate. 30 ms voice frames are used, each consisting of four 7.5 ms subframes (60 16-bit samples), with one subframe look ahead, resulting in total algorithmic delay of 37.5ms. On each of these subframes, 10-th order LPC is applied, and both open-loop and closed-loop pitch period estimation is used on every two frames.

**G.729** ITU-T G.729 [5], based on Conjugate Structure Algebraic Code Excited Linear Prediction (CS-ACELP) algorithm, provides voice encoding at low bitrates (6.4 kb/s, 8 kb/s, and 12.4 kbit/s) at 8kHz sampling rate. The voice frame size is 10 ms and it is divided into two subframes, with a look ahead of one subframe, which means that the total algorithmic delay is 15 ms. 10 ms frame is used for LPC filter coefficients estimation and the 5 ms subframes are used to compute the excitation signal parameters. G.729 supports Voice Activity Detection (VAD) and sends a special Silence Insertion Description (SID) frame, describing the parameters of background noise, when a frame without a voice activity is detected. Since it was designed for use in cellular and network applications, it is also able to interpolate the frames, which are missing due to a transport channel error using a technique called PLC (Packet loss concealment).

**DTX** As stated before, G729 supports different behavior based on whether there is a voice activity in the sent frame - this behavior is called DTX, Discontinuous transmission. First, the encoder determines if there is a voice in a frame and in case there is no voice activity, it either sends a reduced size data packet, which contains the characteristics of the background noise, or no data whatsoever. In that case, the decoder generates the same noise as in previous frames. <sup>5</sup>

**GSM-EFR** ETSI GSM-EFR [3] is a standard for second generation mobile networks, which is used to transport most of today’s mobile communication. Its full rate version operates at 13 kb/s, using Regular Pulse Excitation/Long Term Prediction (RPE/LTP) coder, whereas the enhanced full rate (EFR) uses ACELP codec and at 12.2. kb/s bitrate. The speech frame length is 20ms, each frame consisting of four subframes (5 ms each). Pitch analysis, codebook indices and LPC coefficients are computed for every subframe. Altogether 244 bits of information are sent with every frame, resulting in 12.2kb/s bitrate.

---

<sup>5</sup>[http://www.adaptivedigital.com/pdfspeccs/adt\\_g729.pdf](http://www.adaptivedigital.com/pdfspeccs/adt_g729.pdf)

**Speex** Speex [31] is an open audio compression format based on CELP speech coding algorithm. It is used mainly in VoIP and was designed with a goal of making a codec optimized for high quality speech, working at flexible bit rates. The codec supports ultra-wideband (32kHz), wideband(16 kHz) and narrowband (8kHz) sampling rates and a wide range of bitrates (2-44 kb/s), which can change dynamically depending on the encoded signal. It is possible to choose between variable or constant bitrate mode. Many other advanced features are implemented, including voice activity detection, discontinuous transmission or perceptual enhancement in the decoder.

**MP3** Unlike the others, MPEG-2 Audio Layer III [2], or MP3, is a general purpose codec, which intended use is to compress audio in CD-like quality. The compression ratios for unaltered perceptual quality range in between 9 to 12. Usual CD 2-channel audio has a bitrate of 1411 kb/s at 44.1 kHz sample rate and it is often compressed to a bitrate in range from 128 to 320 kb/s. To achieve such high compression ratio, MP3 exploits combination of perceptual coding techniques and general purpose compression algorithms, like Huffman coding.

Overview of all the codecs used is shown in Table 2.2.

Codec	Bitrates	Bandwidth	Coding techniques	Organization
G.711	64 kb/s	8 kHz	PCM	ITU-T
G.723.1.	5.3, 6.3 kb/s	8 kHz	ACELP, MP-MLQ	ITU-T
G.729	6.4, 8, 11.8 kb/s	8 kHz	CS-ACELP	ITU-T
GSM-EFR	12.2 kb/s	8 kHz	ACELP	ETSI
MP3	8-320 kb/s	8-48 kHz	Many	MPEG
Speex	2-44 kb/s kb/s	8, 16, 32 kHz	CELP	Xiph.Org

Table 2.2: Overview of codecs used in the experimental part

## 2.3 Used features

To classify the codecs, we need to find features that can distinguish them, that have specific behavior for each class. In the following section, some of the more important features, that have proven useful, are shortly discussed.

**MFCC** Mel-Frequency cepstral analysis [13] is one of most widely used techniques of feature extraction in speech processing tasks, typically in speech recognition. Mel-frequency cepstrum coefficients (MFCC) are coefficients obtained through this method. Mel-frequency cepstrum is based on a nonlinear Mel scale, which reflects the human ear scale. The workflow of the algorithm is following: First, an input signal is framed, usually with some overlap and windowed by Hamming window to evade discontinuity and frame edges. Then a fast Fourier transform (FFT) is computed for each frame to obtain frequency components of the signal. Next, the Mel filter bank is applied to the frame processed by FFT. The Mel scale is roughly linear up to 1kHz and then logarithmic. Finally, a discrete cosine transformation (DCT) is performed upon the frame and the resulting coefficients are used as input features.

**LPC coefficients** LPC coefficients are used in LPC based codecs to compress a signal to ensure efficient transmission or storage. It is the prevalent method of compression in medium and low bitrate codecs. The principle of the LPC compression is explained in Section 2.1. Fifteen first LPC coefficients are used in the experiments.

**LPC Residual signal** Also called the error signal, LPC residual signal is the signal which remains after performing an LPC analysis. It is the difference between predicted sample and actual sample. Also see Section 2.1. Residual signals energy, spectral variation and spectral flatness are the features used during the experiments.[30]

**LSF coefficients** Line spectral frequencies (LSF), sometimes also referred to Line spectral pairs(LSP) are a way of representing LPC coefficients during a transmission, for more precise description see for example a paper by Silva comparing speech signal feature extraction methods [28].

## 2.4 Classification

The goal of this work is classification of the codec based on a speech signal, using supervised machine learning classification algorithms. Many techniques were developed during the years a their analysis is well outside the scope of this work. Two approaches were used in the experimental part: SVMs and neural networks. Both are discussed shortly in the next few pages.

### 2.4.1 SVM

Support vector machines (SVM) [6] are a type of an approach to supervised machine learning. They are usually used for classification, however they can also do regression analysis. The algorithm is as follows: First all the samples from two distinct classes (blue and red dots in Figure 2.5) are plotted in  $n$ -th dimensional space, where  $n$  is the dimensionality of our features. Then the space is split in two by a hyper-plane which separates the data-points into two classes as precisely as possible (with as few misclassifications as possible, represented as a solid black line in the figure) and which has the widest possible margin – distance from the hyper-plane to the nearest sample – the space between the solid line and the dotted lines in the figure. If the dataset is not linearly separable, SVMs use a operation called the kernel trick, which transforms the input space into higher dimensional space and makes it separable. In this work, linear SVM is used with an one-vs-rest strategy to transform the multiclass classification into binary classification. That means there are no kernel transformations involved and one model is built for each class, where the samples belonging in this class are treated as a positive result and all the other samples as a negative result.

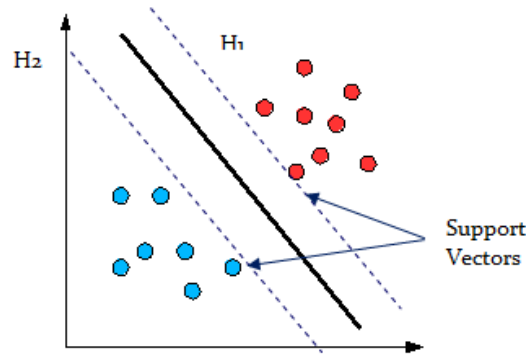


Figure 2.5: Principle of SVM<sup>6</sup>

### 2.4.2 Neural Networks

Artificial neural networks are computational models loosely inspired by human brain function. They learn to produce an output based on input data by learning on large number of samples. The networks consist of number of interconnected nodes, also called neurons, which work together to solve a given problem. This problem is usually either output prediction, pattern recognition, or, as is the case in this thesis, classification. The neurons are organized in layers, typically there is one input layer, one output layer and one or more layers in between them - these are called the hidden layers and number of hidden layers defines a *depth* of the network. The samples are fed into the network via input layer. The actual computations are done by the neurons inside hidden layers. One of the simplest and oldest neuron models, called the perceptron, consists of one or more inputs  $x_1, x_2, \dots, x_n$  that are multiplied by weights  $w_1, w_2, \dots, w_n$  and one binary output  $o$ . This output is set to one if the sum of the weighted inputs exceeds a threshold  $t$ , i.e. :

$$o = \begin{cases} 1 & \text{for } \sum_{i=1}^n x_i w_i > t \\ 0 & \text{for } \sum_{i=1}^n x_i w_i \leq t \end{cases} \quad (2.1)$$

The function deciding an output of a neuron is often referred to as an *activation function*. Nowadays, more complex functions than a threshold are employed, since it is very hard to train perceptron network with thresholded binary output - it is very sensitive, even a slight change on the input might flip the state of a neuron from 0 to 1 and change the overall output in almost unpredictable way. The popular choices of activation function are sigmoid, tanh or ReLU, resulting in real-valued inputs and outputs.

**Sigmoid** Sigmoid takes an input and 'squashes' it into range between 0 and 1. It was historically one of the most used activation functions, since it provides a nice biological analogy of a neuron function - 0 means inactive neuron and 1 represents a neuron firing at full frequency. Nowadays it is rarely used in practice due to its many drawbacks, most inconvenient of them being the saturation problem. The gradient at the minimum and maximum of the function (0 and 1) are almost zero - this means there will be very little

<sup>6</sup>The figure comes from an online course STAT 897D by PennState university <https://onlinecourses.science.psu.edu/stat857/node/240>



gradient flowing through the neuron during backpropagation and it will be very difficult to train the network. The equation of a sigmoid function looks like this:

$$\sigma(x) = \frac{1}{(1 + e^{-x})} \quad (2.2)$$

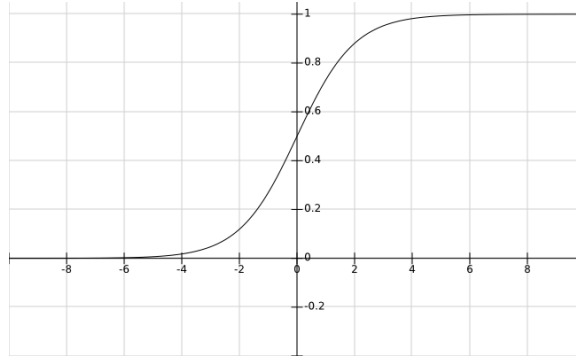


Figure 2.6: sigmoid function

**Hyperbolic tangent** Tanh is a non-linearity similar to a sigmoid, in fact, it is just a scaled sigmoid:

$$\tanh(x) = 2\sigma(2x) - 1 \quad (2.3)$$

It squashes the input into range  $(-1,1)$  and its principal advantage compared to sigmoid is that the output is zero-centered.

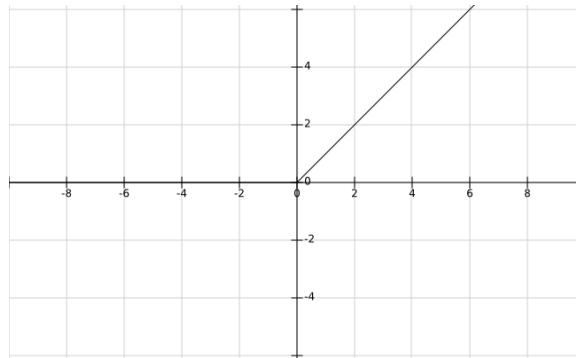


Figure 2.7: Hyperbolic tangent function

**ReLU** Rectifier linear unit [15], shown in Figure 2.8, simply outputs zero for negative inputs and lets through any positive input unchanged. ReLU has grown very popular in last years, since the computation is very fast and research has shown it may greatly speed up stochastic gradient descent convergence, documented for example by A. Krizhevsky and I. Sutskever [22].

$$f(x) = \max(0, x) \quad (2.4)$$

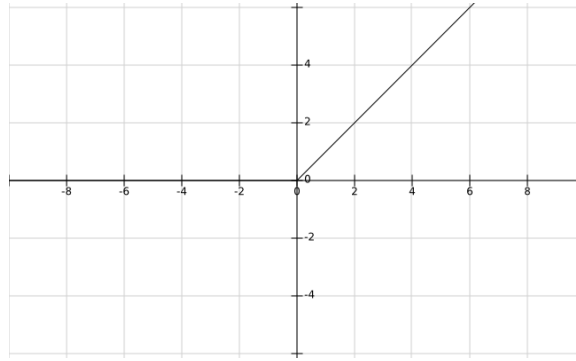


Figure 2.8: ReLU function

The architecture described above can be used as an universal computational model, as proven by G. Cybenko[12], but it does not learn anything. In neural networks, learning means simply automatically adjusting the input weights to minimize the error function of the output. In the training data, each input  $x_i$  is paired with a label  $y_i$ . The error function is computed based on difference of the label  $y_i$  and the network output  $o_i$ . The goal is to match the output  $o_i$  with the corresponding label  $y_i$  as closely as possible. In other words, if  $f(x)$  is a function that produces  $y_i$  for every  $x_i$ , we are trying to approximate this function by  $f^*(x)$ , that is composed by functions of the individual layers:  $f^*(x_i) = f_n(f_2(f_1(x_1)))$ . The actual adjustment of the weights is usually done via the backpropagation algorithm in combination with some modification of gradient descent. The gradient for each weight is computed based on the output of an error (loss) function, the weight is modified and the network output converges towards an optimum.

### 2.4.3 Feedforward neural networks

Feedforward neural networks (also referred to as multi layer perceptron networks, MLPs)[14] are networks where the information is always propagated in one direction - there are no loops in connections between the nodes. In other words, the network can be represented as an acyclic oriented graph connecting the nodes, which perform an operation defined by their activation function on their inputs<sup>7</sup>.

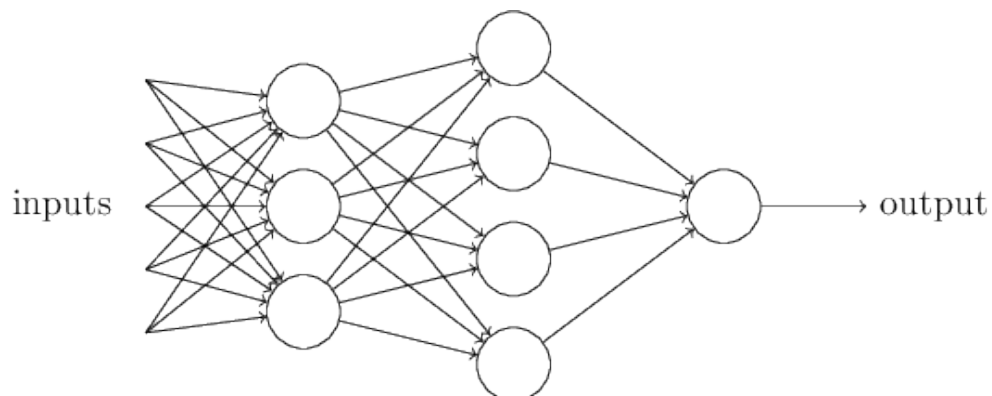


Figure 2.9: Principle of a feedforward neural network<sup>8</sup>

<sup>7</sup><http://www.deeplearningbook.org/contents/mlp.html>

#### 2.4.4 Recurrent neural networks

Recurrent neural networks (RNNs) are designed to make use of sequential data, which is exactly what a framed signal is. The connections between units make up a directed cycle, which allows the network to pass an information from previous state to the current one, as illustrated in Figure 2.10. In other words, hold a state based on previous inputs. In a feedforward network, each input is processed separately, without any influence of previous samples. This ability to process sequences of data should prove useful in the task of signal processing. One of the most serious problems of classical RNNs, is the vanishing gradient problem. The traditional activation functions have gradients in range  $(-1, 1)$ , causing the error gradient to vanish exponentially with each timestep. This problem renders the classic RNNs unusable for analyzing data with long-term dependencies, as shown by Yoshua Bengio[7].

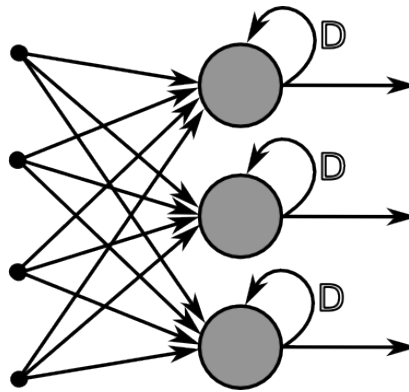


Figure 2.10: A scheme of a recurrent neural network<sup>9</sup>

#### LSTM

LSTM (Long short-term memory) neural networks are a modification of a basic RNN model proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber[17]. They reduce the vanishing gradient problem, which makes them much more effective on capturing long-term dependencies. The key of the solution is usage of multiple gates and a cell state, which runs through all the cells and is manipulated using these gates – parts of the state may be added or removed. This enables the network to read, forget and write selectively, preserve the error and backpropagate it into much deeper layers. Each gate is in fact a sigmoid layer that outputs a number between 0 and 1. This number represents the degree of how much the cell state is modified. The first gate is the forget gate, which says how much of the state is going to be dropped. Then we need to know how much of the results of current iteration to keep – input gate serves this purpose, it allows the network to “pick” the part of the current result (also called the candidate result) to combine with a cell state. The last gate, the output one, controls how much of the information flows out from the cell and is made exposed to other cells and next timesteps. By tuning the weights for these gates, the network learns how to operate with its own state.

<sup>8</sup>Figure from Michael Nielsen’s book, <http://neuralnetworksanddeeplearning.com/chap1.html>

<sup>9</sup><https://github.com/cazala/synaptic/wiki/Neural-Networks-101>

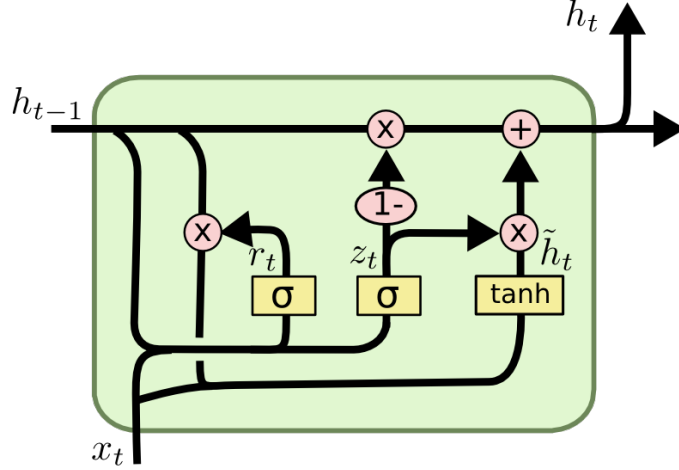


Figure 2.11: Structure of LSTM cell<sup>10</sup>

The step by step workflow of LSTM network is the following, based at Michael Nielsen's book [24] and an article by Christopher Olah [25]. Let us assume that in time  $t$  we have an input vector  $\mathbf{x}_t$ , an inner cell state  $\mathbf{c}_{t-1}$  and an output from previous step,  $\mathbf{h}_{t-1}$ , which is also called hidden state. First, the network decides how much of the information from previous steps to keep stored in its cell state. This decision is done by the forget gate, which consists of a sigmoid function applied to weighted sum of previous output, input plus a bias:

$$\mathbf{f}_t = \sigma(\mathbf{W}_{\mathbf{x}\mathbf{f}}\mathbf{x}_t + \mathbf{W}_{\mathbf{h}\mathbf{f}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathbf{f}}) \quad (2.5)$$

In Equation 2.5,  $\mathbf{W}$  are the weights, which are updated through the backpropagation algorithm and  $\mathbf{b}_{\mathbf{f}}$  is the bias, which is habitually set at 1 (as well as in experiments in this paper) and improves the networks performance as shown by Rafal Jozefowicz's paper [20]. As apparent from the equation, the forget gate in fact states how much of the information to remember, i.e. value 1 means to keep everything stored in the cell state, which may be a little confusing at first.

The subsequent step is to decide how much of the inner state is going to be updated, i.e. which portion of the result is the cell going to store in its state. The input gate serves this purpose. Its equation is very similar to the one before:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{\mathbf{x}\mathbf{i}}\mathbf{x}_t + \mathbf{W}_{\mathbf{h}\mathbf{i}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathbf{i}}) \quad (2.6)$$

At this point, we know in what way is the state going to be modified, but we still need to compute the new values which will be stored in it. These values are often called the *candidate values*. They are computed as an activation function (hyperbolic tangent is a popular choice) of the weighted sum of input and previous output plus bias:

$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{\mathbf{x}\mathbf{c}}\mathbf{x}_t + \mathbf{W}_{\mathbf{h}\mathbf{c}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathbf{c}}) \quad (2.7)$$

Now it is possible to update the cell state based on the two gates, previous state and the candidate values. The procedure is as follows. We multiply the previous state by the

<sup>10</sup>Figure from <http://colah.github.io/posts/2015-08-Understanding-LSTMs/>, a site with very comprehensible and intuitive explanations of machine learning topics

forget gate, the candidate values by the input gate and add these two vectors together to obtain a new cell state for the current timestep,  $\mathbf{c}_t$ :

$$\mathbf{c}_t = \tilde{\mathbf{c}}_t \odot \mathbf{i}_t + \mathbf{c}_{t-1} \odot \mathbf{f}_t \quad (2.8)$$

The only step that remains is to produce the actual output of the cell. This output is based on the newly updated cell state, which is transformed by tanh function and multiplied by the output gate, which has a familiar form of:

$$\mathbf{o}_t = \sigma(\mathbf{W}_{xo}\mathbf{x}_t + \mathbf{W}_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \quad (2.9)$$

And the final equation producing the result (and the hidden state for the next timestep) is:

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh \mathbf{c}_t \quad (2.10)$$

## GRU

Gated Recurrent Unit (GRU) networks, introduced by Cho et al. [11] are very similar to LSTM networks and have been found to yield comparable results on various machine learning tasks, see for example a paper by Cho et al.[10]. The principal difference is that the output gate is omitted, so the GRU only uses two gates, which results in less matrix multiplications and thus better performance in terms of computational complexity. These gates are called *reset* and *update gate*. Update gate translates roughly to forget and input gates of a LSTM and reset gate modifies if the output hidden state is based more on the current input, or the previous hidden state. Also, the hidden state and the cell state are merged into one, and there is no second nonlinearity applied to the hidden state (like tanh in (2.1)). This network architecture was proposed in 2014 by Cho et al. [11]. The workflow of the unit is very similar to LSTM. The units hidden state is updated based on the update gate, which has an equation:

$$\mathbf{z}_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1}) \quad (2.11)$$

Analogically, the reset gate:

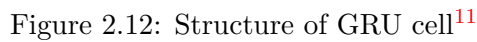
$$\mathbf{r}_t = \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1}) \quad (2.12)$$

The new candidate hidden state is now, after applying reset gate:

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{r} \odot \mathbf{W}_{hh}\mathbf{h}_{t-1}) \quad (2.13)$$

And the final state of the cell, combining the candidate output and the previous hidden state is:

$$\mathbf{h}_t = (\mathbf{1} - \mathbf{z}_t)\mathbf{h}_{t-1} + \tilde{\mathbf{h}}_t\mathbf{z}_t \quad (2.14)$$



step to take. This parameter is called the learning rate and it is one of the most important hyperparameters to tune in neural networks.

**SGD** The difference between ordinary gradient descent and stochastic gradient descent (SGD) is that ordinary gradient descent is computed over the whole dataset, whereas SGD updates the parameters for every sample and target pair. This has few advantages and few disadvantages. Since the data are usually correlated, it is redundant, even wasteful to compute the objective function for the whole dataset to make a single parameter update. We also must be able to fit all the data into memory in order to compute the function output. On the other hand, when we compute the function for a single sample, the variance is greater and this fluctuation may make it harder to reach a local or global minimum, since we may overstep it. In practice, SGD is usually used to denominate mini-batch SGD, which takes the best from both approaches – the update is done over a small subset of data.

**Adam** Adaptive Moment Estimation (Adam) [21] is a modified version of SGD, where the learning rate is set and changed automatically throughout the learning process.

## Regularization

Since the ultimate goal of machine learning is to be able to perform on unseen data, its ability to generalize the patterns it has learned is crucial. One of the pitfalls of neural network classification is called **overfitting**. During the training period, the model fits its weights to the training data. The overfitting problem arises when the model generates very complex functions to fit the training set almost perfectly, when it has a lot of parameters compared to number of training samples. In that case, it will be very difficult for the network to generalize, since even a small fluctuation, or a random noise in input data can influence the output. On the other hand, a model that is too simple does not fit even the training data, so it is even less useful. There are many techniques designed to find the compromise between the two extremes, a model that fits the training set well, but that keeps low enough complexity to generalize reasonably. These techniques are collectively referred to as regularization. Regularization does not lower training error, it fact often even raises it, but it is intended to lower the *test* error.

The most used methods in neural networks are the L1 and L2 regularization and dropout [29]. The idea behind L1 and L2 regularization is to penalize large model weights, so that the model does not fit training data perfectly. The implementation of both is straightforward: we add one more term to the loss function. This term represents the penalty and the way it is computed is the difference between L1 and L2 regularization. In case of L1 regularization, the term is computed as a sum of absolute values of all weights, whereas in L2 regularization, it is sum over squared values of all weights. The equations are following:

$$C^* = C + \frac{\lambda}{n} \sum_w |w| \quad (2.15)$$

$$C^* = C + \frac{\lambda}{2n} \sum_w w^2 \quad (2.16)$$

where  $C$  is the original cost function,  $C^*$  is the new cost function,  $w$  is the input weight,  $n$  is the number of input samples and finally  $\lambda$  is the coefficient, that is usually tuned using

cross-validation. Thus, models with lower weights are favored, which intuitively means the model will not be affected too much by random noise in the input and will learn more general patterns than unregularized one.

**Dropout** The idea behind dropout is fundamentally different, simpler, but the goal is similar. Dropout randomly selects predefined fraction of input connections for each data sample and sets the according weights to 0, i.e. turns a portion of the neurons off. Neurons are only switched off during the training phase. This makes the model more robust, since it is forced to learn to produce output based on different subset of features. One way to look at why dropout helps to solve overfitting is that in fact, with each change of dropped neurons, we train a slightly different network, which overfits in different way. Then all of the models are averaged, when we turn the dropout off in testing and actually using the model for predictions. One of the pitfalls of conventional dropout is that it does not perform very well on RNNs – simply put, dropout on the recurrent connection „corrupts“ the ability of the network to learn long term dependencies. However, there are modified algorithms that do help to decrease overfitting even in recurrent nets, for example a paper by Zaremba, Sutskever and Vinyals [32].

#### 2.4.5 Metrics

In the next two paragraphs, I will discuss the classification metrics that will be used during the experiments on an example with binary classification, where we have data in two classes and the classifier predicts whether the input belongs to one of them, i.e. the classifier outputs positive prediction if the sample is classified into the first class, and a negative prediction if the sample is classified into the second class. This explanation can be generalized to arbitrary number of classes. To explain the metrics, four terms must be defined:

**True positives (TP)** - Number of times the classifier predicted positive result and the input indeed belonged into the first class.

**True negatives (TN)** - Number of times the classifier predicted negative result and the input was in the second class.

**False positives (FP)** - Number of times the classifier predicted positive result, but the input belonged into the second class.

**False negatives (FN)** - Number of times the classifier predicted negative result, but the input belonged into the first class.

#### Accuracy

Accuracy is the most simple and most commonly used evaluation metric for classification problems. It is the proportion of the true negatives and true positives among the classifier outputs, that means it is the number of times the classifier was correct, divided by the total number of classifications.



## F1 score

The principal metric used in the experiments is F1 score, also called the F1-measure:

$$F = 2 \times \frac{p \times r}{p + r} \quad (2.17)$$

It is a harmonic mean between the precision and recall. Harmonic mean can be viewed as a very conservative average – the result is closer to a minimum of the two values than in case of using geometric mean. Precision  $p$  is the number of true positives divided by the sum of true positives and false positives:

$$p = \frac{TP}{TP + FP} \quad (2.18)$$

and recall  $r$  is the number of true positives divided by the sum of true positives and false negatives:

$$r = \frac{TP}{TP + FN} \quad (2.19)$$

To show what precision and recall is on an example, I consider a set of samples, where half of them has a value A and half of them has a value B. Then we take the samples which were classified as A, and see which part of them had in fact value A. This ratio is the precision. Now we see how many samples were classified as A correctly and what is the real count of A samples in the dataset. Ratio of these two numbers is the recall. This approach works only for two classes – one considered a positive and the other one a negative result. However, in practice, as well as in this work, the results of precision and recall for all the possible binary combinations are represented as a confusion matrices, which are averaged in order to yield a single number for all the classes.

## Comparison

F1 score should almost always be preferred to simple accuracy, especially in cases with a strong class imbalance. In this work, F1 and accuracy are very similar, since the all the classes are equally sized and in most of the experiment, both precision and recall are high, above 0.9, and very similar, so their harmonic mean is also above 0.9 and corresponds to accuracy. Therefore, the main motivation for using F1-score is because this metric is universally accepted and it is possible to compare the results directly with other papers.

## Chapter 3

# Data

The speech corpus used for experiments consists of short (2-20 seconds) audio files downloaded from [voxforge.org](http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/16kHz_16bit/cmu_com_kal_ldom.tgz), namely corpora in `cmu_com_kal_ldom`<sup>1</sup>, `cmu_us_awb_arctic`<sup>2</sup> and `8kHz`<sup>3</sup> directories. Each file typically contains one sentence. The database contains recording from many speakers with different accents and dialects of English language, recorded at a different recording qualities. The first part (473 files) of the training/validation data used was recorded with 8 kHz sampling rate and 16 bit sample size on a telephone microphone, the second part (1304) was recorded with 16 kHz sampling rate and 16 bit sample size. The test data come from corpus CMU\_ARCTIC US uwb, which consists of 1138 utterances from Project Gutenberg by a Scottish male, recorded at 16 kHz. First, a bash script was used to resample all the files to 8 kHz as some of the codecs (G.711, G.723, G.729, GSM-HR) require this sampling rate, encode them with desired codec and then decode them back into a raw wave file. A bash script using ffmpeg was used to encode all the files. The files were randomly shuffled and partitioned into training (700 files for each codec), validation (100 for each codec) and test (again 100 files) sets. Validation set was used for hyperparameter and architecture tuning, whereas the test set was used for the final evaluation.

Set	Size	Original corpus
Training	700	8 kHz <sup>1</sup> and <code>cmu_com_kal_ldom</code> <sup>2</sup>
Validation	100	8 kHz <sup>1</sup> and <code>cmu_com_kal_ldom</code> <sup>2</sup>
Test	100	CMU_ARCTIC US uwb <sup>3</sup>

Table 3.1: Used sets

---

<sup>1</sup> [http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/16kHz\\_16bit/cmu\\_com\\_kal\\_ldom.tgz](http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/16kHz_16bit/cmu_com_kal_ldom.tgz)

<sup>2</sup> [http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/16kHz\\_16bit/cmu\\_us\\_awb\\_arctic.tgz](http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/16kHz_16bit/cmu_us_awb_arctic.tgz)

<sup>3</sup> [http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/8kHz\\_16bit/](http://www.repository.voxforge1.org/downloads/SpeechCorpus/Trunk/Audio/Original/8kHz_16bit/)

<b>Codec</b>	Bitrate	Other settings
G.711	64 kb/s	
G.723.1	6.3 kb/s	
G.729	8 kb/s	DTX on
GSM-EFR	12.2 kb/s	DTX on
MP3	8 kb/s	
Speex	variable (ffmpeg default)	

Table 3.2: Configuration of encoders used in the experimental part

## Chapter 4

# Implementation

In the previous section, all the components necessary for the experiments were outlined in theory. In the next sections, I will describe how the particular parts were implemented and which tools I used to develop a working experimental framework, from preprocessing the data, through extracting the features, to finally classify the files. An overview of the used tools can be seen in Figure 4.1, which defines the tools used in each step of the pipeline from Figure 2.1.

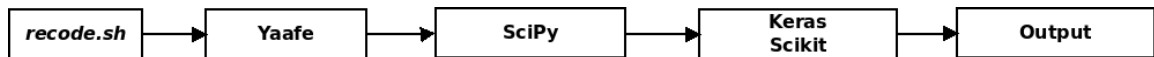


Figure 4.1: Tools used in the classification pipeline

### 4.1 Used tools

As apparent from Figure 4.1, a number of libraries and tools was used to prepare the data and carry out the experiments. This section provides an introduction to each of these tools.

#### 4.1.1 Soundfile

Soundfile is an audio library based on libsndfile that allows us to open a sound file and store the signal levels in a NumPy array, which is a format used by all the other components of the system. File format, sample rate and sample size can be deduced from a header or set explicitly.

#### 4.1.2 Yaafe

Yet Another Audio Feature Extractor – Yaafe[23] is a feature extraction software written in C++ and Python, which offers many advantages over its competitors, at least in open-source community. It provides a big range of available features while keeping low computation complexity due to exploiting feature computation redundancies. In other words, many of the features share the same intermediate representations and parts of the algorithm, like FFT, signal envelope or spectrum magnitude. These representations are computed only once for each signal and then they are used repeatedly for each feature extraction algorithm that needs them. To allow this behavior, Yaafe works in two steps. The first step is to create a feature extraction plan. A feature extraction plan is a Python class, whose

attributes can be set manually or can be read from a file. Each line of such file defines one feature to extract, e.g.

---

```
lpc:LPC LPCNbCoeffs=10 blockSize=256 stepSize=128
```

---

This line will tell the engine to split the signal into frames of 256 samples with 50% overlap and compute the first ten LPC coefficients for each frame. Each feature is represented as a series of steps, for example Frames, FFT, MelFilterBank and Cepstrum for Mel-frequency cepstral coefficients. The feature plan is then parsed into a reduced directed graph with all the redundancies omitted, because each of the steps is computed only once and the results are shared between different features. Yaafe can be used as a command line program with the results stored in HD5F or CSV file format, or it can be imported as library in a Python script and in this case, the results are returned as a NumPy array, which makes it easy to process them by the next tools, SciPy[19] and Keras[9].

### 4.1.3 SciPy

SciPy is a collection of functions and algorithms extending the NumPy library. It also provides the user with ability to plot graphs bringing Python close to tools like MATLAB or Octave. In this work, SciPy used for data preprocessing.

### 4.1.4 Keras

Keras[9] is a high-level Python neural networks library. Its development goals are simplicity of prototyping while keeping maximal flexibility of the network. The computational engine can run on top of either Theano or Tensorflow deep learning libraries. Keras code can run on either CPU or GPU and the benchmarks show that it is one of the fastest Python neural network libraries available.

There are two ways of defining a model in Keras. The first, more straightforward, is the Sequential API, where you initialize a class called Sequential and then linearly add layers via the add method, for example:

---

```
model = Sequential()
model.add(Dense(64, activation='relu', input_dim=50))
```

---

where Dense is a name of the class representing the layer, activation is name of the activation function, 64 is the output dimension and 50 is the input dimension (number of features). Dense layer is just a regular fully connected network layer where the output of each node is connected to the inputs of all nodes in a consecutive layer.

Another way, which is used in this work, is so called functional model, which allows multi-input and multi-output networks, shared layers and generally gives more flexibility to the developer. The idea is following – all layers are in fact functions, that take an input in form of a tensor, transform it and return a tensor again. Thus, layers are callable and accept output of a different layer as an input. Also the whole model is callable in exactly the same way, which makes it easy to embed a complete, possibly pre-trained model into a more complex structure. This work exploits this feature by combining the feedforward statistical model and recurrent model, concatenating the outputs of the last hidden layers of both and adding a dense layer on top of them. It is also possible to share a layer between different models, or train part of the model with different weights. Example:

---

```
g=GRU(128,return_sequences=True) (input1)
g=GRU(256,dropout_W=0.2,return_sequences=True) (g)
```

---

The first layer shown takes an output tensor from layer `input1` and transforms it (GRU is the type of recurrent network used in this work) . The next layer does the same operation on the output tensor of the first layer – more detailed description of what exactly is happening here can be found in the implementation part of this work.

After we are done with architecture definition of the model, we need to configure how to compute loss, backpropagate it in the model to adjust weights and how to evaluate the results – define the metrics used. All three things can be done via the 'compile' method:

```
model.compile(loss='categorical_crossentropy', optimizer=SGD(lr=0.01),
              metrics=['accuracy'])
```

---

This line sets categorical crossentropy as the loss function (function, which output are we trying to minimize during the training) and uses stochastic gradient descent with a learning rate of 0.01 to propagate the loss. Metrics values are shown in a text output during the training and evaluation. There are more ways to feed the data into the model, in this work, fit method of the Model class is used.

```
model.fit([x[:-200],labels[:-200]], batch_size=40,
          validation_data=[x[-200:],labels[-200:]], nb_epoch=50)
```

---

This line says that training data and their labels are stored in variables *x* and *labels*. For the training phase, use all of them, except for the last 200, process data in batches of size 40 – this means that the loss is computed and propagated through SGD and backpropagation every 40 samples – validation data are the 200 last elements in *x* and *labels*, and we will train for 50 iterations over the whole dataset. After training, it is possible to save the weights of the network in HDF5 format via

```
model.save_weights(filename)
```

---

To show a concrete example of a network generated by Keras, consider the following source code, which is very similar to the actual code of the network used in the experiments. A diagram of such network can be seen in Figure 4.2 and a graphic representation of a single neuron in Figure 4.3.

```
input=Input(shape=(3,))
d=Dense(6,activation="relu")(d)
d=Dense(6,activation="relu")(d)
d=Dense(6,activation="relu")(d)
output=Dense(2, activation='softmax')(d)
model=Model(input=input2,output=output2)
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(clipnorm=1.),
              metrics=['fbeta_score'])
```

---

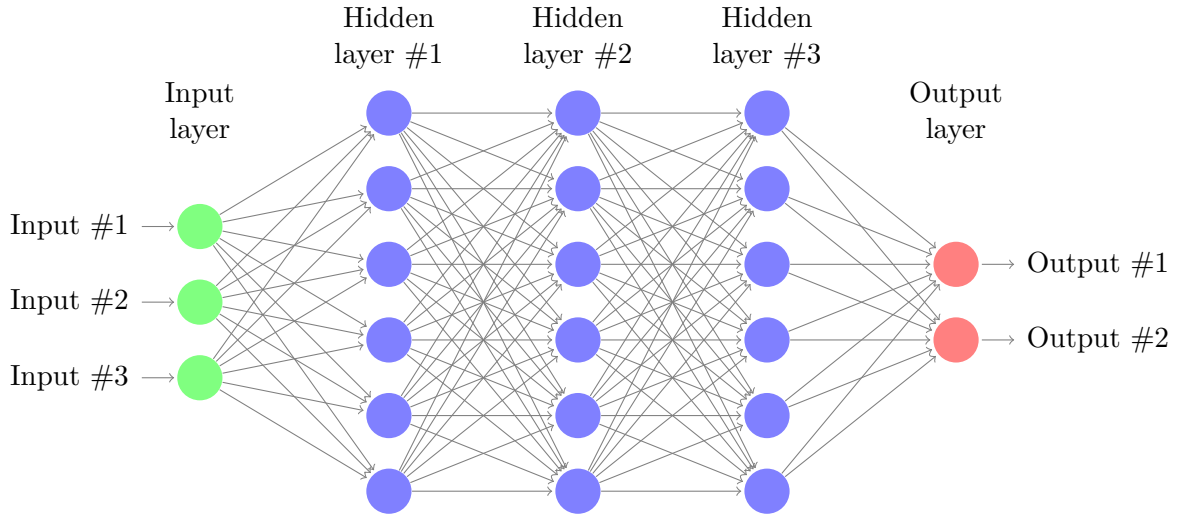


Figure 4.2: A feedforward neural network with three hidden layers

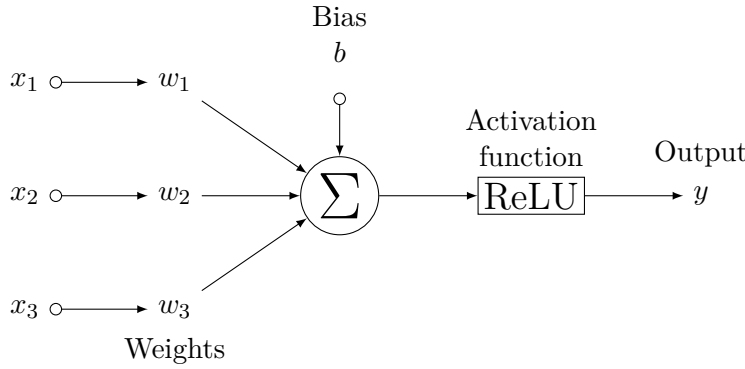


Figure 4.3: A single ReLU neuron <sup>1</sup>

## 4.2 Data preprocessing

After downloading the data, script *recode.sh* was used to create the directory structure, resample and encode the input files with used codecs. The scripts takes the original input file, recodes it into desired format and then into uncompressed sound file with sample size of 16 bits and 8 kHz sample rate.

## 4.3 Feature extraction

Now that the corpus is prepared, feature extraction can be executed upon it. LPC error signal, sometimes also called the residual signal, was extracted from the files using Edinburgh Speech Tools. <sup>2</sup> and saved in an auxiliary file. Then each file was loaded into NumPy

<sup>1</sup><http://tex.stackexchange.com/questions/132444/diagram-of-an-artificial-neural-network>

<sup>2</sup>[http://www.cstr.ed.ac.uk/projects/speech\\_tools/](http://www.cstr.ed.ac.uk/projects/speech_tools/)

array using Soundfile library and the features were extracted from both the original signal and the residual signal with a Python script, using SciPy, NumPy and Yaafe libraries.

In a speech processing, the signal is usually split into 10 – 40 ms long frames. 256 samples at 8000 kHz sampling rate are equal to 32 ms and as you can see in the experimental part, this size has proven to be ideal. However, for the baseline experiments, resulting NumPy array was split into frames of 1024 samples with 50% overlap. From these frames, features were extracted using Yaafe. Summary of all features used during the baseline experiments can be seen in Table 4.1:

Feature	Coefficients	Frame size	Step size	Other parameters
Autocorrelation	15	1024	512	-
Hilbert envelope	1	1024	512	-
Energy	1	1024	512	-
Envelope shape statistics	1	1024	512	<sup>3</sup>
Spectral flatness	1	1024	512	-
Spectral variation	1	1024	512	-
Spectral shape statistics	1	1024	512	<sup>4</sup>
Zero crossing rate	1	1024	512	-
MFCC	13	1024	512	-
LPC coefficients	15	1024	512	-
LSF coefficients	15	1024	512	-
LPC residual. energy	1	1024	512	-
LPC res. spect. variation	1	1024	512	-
LPC res. spect. flatness	1	1024	512	-

Table 4.1: Summary of used baseline features

### 4.3.1 Feature statistical description

For the feedforward network architecture used later, 4 statistical functions were used on the features: skewness, kurtosis, mean and variance. All of them are part of the stats module from SciPy library.

## 4.4 Classifiers

After obtaining the features and scaling them using SciPy, it is finally possible to classify the samples.

### 4.4.1 SVM

A module called LinearSVC (Support vector classifier) from scikit-learn library was used as a simple baseline. It employs a linear kernel and one-vs.-rest strategy to handle multiclass classification. LinearSVC expects the input to have zero mean a unit variance – function scale from SciPy library was used to accomplish this task.

<sup>4</sup>Centroid, spread, skewness and kurtosis

<sup>4</sup>Same as above



---

```
from sklearn import preprocessing  
X_scaled = preprocessing.scale(X)
```

---

#### 4.4.2 Neural networks

Keras was used to build models of the neural networks. In the baseline experiments, three networks were used: feedforward network, recurrent GRU network and a combination of the above. Exact specification of the networks is presented in the next chapter.

## Chapter 5

# Experiments

### 5.1 Hardware

Unless stated otherwise, the experiments were conducted on a laptop with Intel Core i7-2620M CPU, 4GB RAM and SSD hard drive. The final experiments, using the best networks, with increased amounts of data, were performed on Czech computer grid for academic community, Metacentrum<sup>1</sup>.

### 5.2 Network architectures

Three network models were designed – feedforward, recurrent and combined. Categorical crossentropy was used as a loss function, Adam as an optimizer and F1-score as an evaluation metric. More detailed description of these functions can be found in the Theory in Chapter 2. For the baseline experiments, the architectures of the networks were the following (corresponding Keras codes are shown). The feedforward network had one hidden fully connected layer with 256 neurons, rectifier linear unit as an activation function and an output layer with 6 neurons activated by a softmax function.

---

```
input2=Input(shape=(x2.shape[1],))
d=Dense(FEEDFORWARD_NEURONS,activation="relu")(b)
output2=Dense(classes, activation='softmax')(d)
model2=Model(input=input2,output=output2)
model2.compile(loss='categorical_crossentropy',
               optimizer=Adam(clipnorm=1.),
               metrics=['fbeta_score','acc'])
```

---

The recurrent network was built using a GRU layer, with 256 nodes activated by hyperbolic tangent activation function and the same output layer as above. The first 150 feature frames were used as an input.

---

```
input1=Input(shape=(FRAMES,x.shape[2]))
g=GRU(RECURRENT_NEURONS,activation="tanh",return_sequences=False)(input1)g)
output1=Dense(classes, activation='softmax')(g)
model=Model(input=input1,output=output1)
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
```

---

<sup>1</sup><http://www.metacentrum.cz>

---

```
metrics=['fbeta_score', 'acc'])
```

---

Also a linear SVM was used for a comparison:

---

```
clf = svm.LinearSVC(C=10)
cf=clf.fit(x2[:-TEST_FILES_COUNT], labels.argmax(1)[: -TEST_FILES_COUNT])
pred=clf.predict(x2[-TEST_FILES_COUNT:])
print (metrics.f1_score(labels.argmax(1)[-TEST_FILES_COUNT:], pred,
    average='weighted'))
print (metrics.accuracy_score(labels.argmax(1)[-TEST_FILES_COUNT:], pred))
```

---

### 5.3 Data

For the baseline experiment, 4800 (800 for each codec) files from training and validation datasets were used, 4200 for training and another 600 (randomly chosen from the whole dataset) files were set aside for validation. Samples were randomly shuffled before running the training.

### 5.4 Features

Features used in this work were selected based on previous works by Sharma et al. [27] and F. Jenner and H. Kwasinski [18]. For the feedforward network, 4 statistical functions were calculated for each feature – mean, variance, skewness and kurtosis. Input of the recurrent network consisted of the first 150 frames for each feature. In the baseline tests, silent frames were still present in the dataset. To speed up the experiments, preprocessed features were saved in a binary format using *pickle* library distributed with Python. For the baseline tests, features described in Table 4.1 were used with a frame size of 1024 samples and 50% overlap, resulting in total of 74 features.

### 5.5 Baseline results

Exact specifications of the classifiers and results are presented in Table 5.1

Parameter	Feedforward	GRU	Combined	SVM
Layers	1	1	3	-
Neurons per layer	256	256	256	-
Activation	ReLU	tanh	ReLU / tanh	-
Regularization	None	None	None	-
Optimizer	Adam	Adam	Adam	-
Loss function	Cat. crossentropy	Cat. crossentropy	Cat. crossentropy	-
Batch size	80	80	80	-
Frame size	1024	1024	1024	1024
# of frames	-	150	150	-
# of epochs	150	50	50	-
Epoch duration	0.4s	123s	125s	-
F1 score	<b>0.9358</b>	<b>0.9406</b>	<b>0.9309</b>	<b>0.8609</b>

Table 5.1: Comparison of baseline classifiers

Both models yielded similar results, but the simpler feedforward network converged much more quickly. The feedforward network was trained during 150 training epochs (iterations over the whole dataset), while the recurrent network was trained for 50 epochs. The duration of one epoch was around 0.4 s for the feedforward network and 123 s for the recurrent one. Peak memory usage was around 600 MB for the feedforward network and 900 MB for the recurrent one. Courses of the F1-score function on evaluation set during the training can be seen in figure 5.2. The Y axis represents the score, while the X axis shows the number of epochs. The comparison of the classifier architectures used is shown in Figure 5.1 and Table 5.1. It is obvious that the combined architecture does not bring any improvement compared to recurrent model, and only makes the model more difficult to train. Based on this observation, only the two distinct neural network architectures and a SVM classifier will be used in the following experiments.

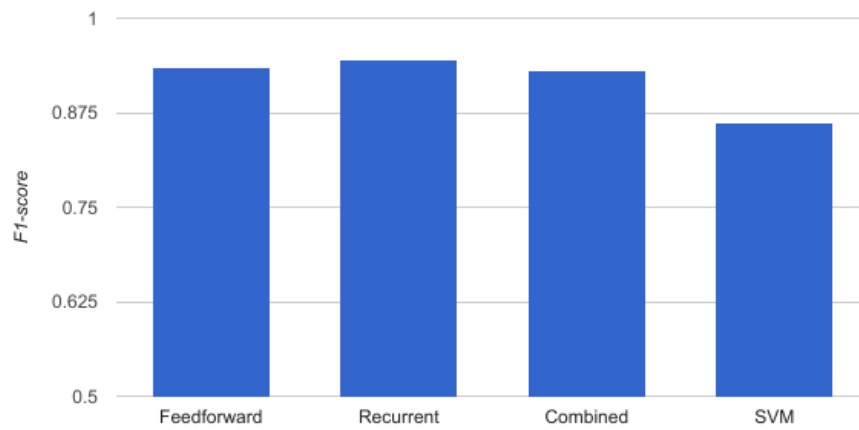
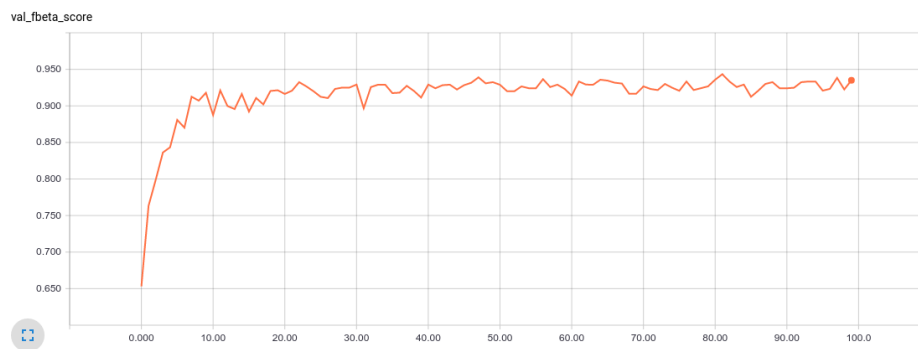
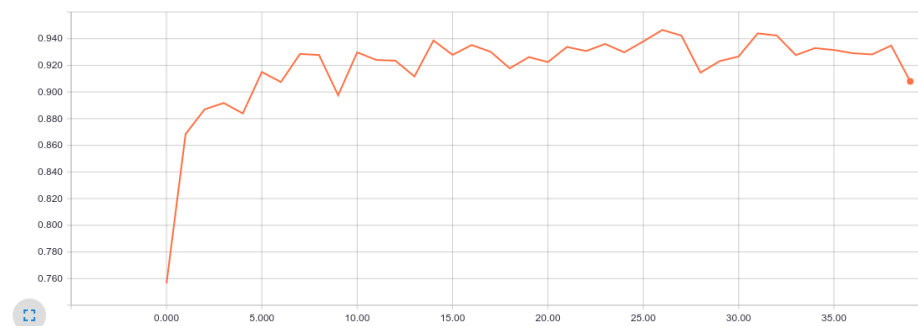


Figure 5.1: Comparison of baseline classification methods.



(a) Feedforward network



(b) Recurrent network

Figure 5.2: Learning curve of the models in the baseline setup.

## 5.6 Improving the networks

These baseline results are consistent with studied research papers referenced in the introduction. Still, there is a number of steps we can take to improve the accuracy.

### 5.6.1 Possible improvements

Possible ways of improving the models performance include:

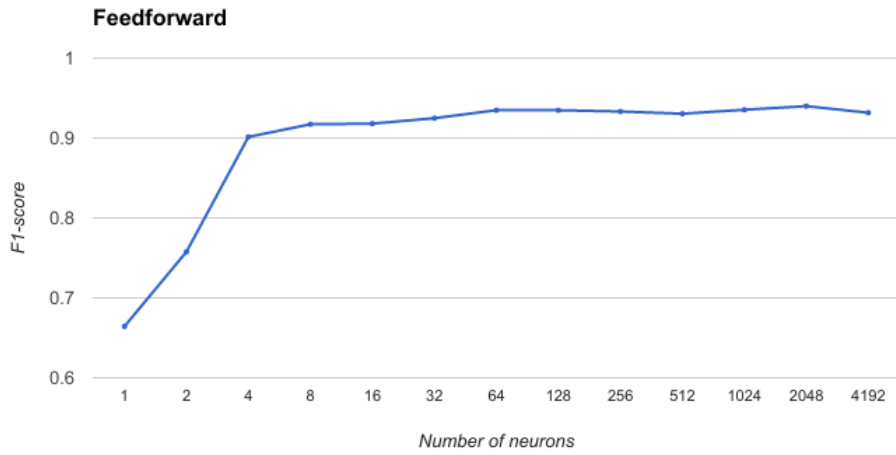
- different activation functions – tanh, sigmoid, leaky ReLU
- different optimizer – RMSprop, adagrad
- different network architecture (using LSTM instead of GRU, Highway instead of Dense)
- data normalization
- regularization techniques (e.g. dropouts to prevent overfitting)
- tuning the hyperparameters
  - number of layers
  - number of nodes in each layer
  - learning rate, momentum, decay rate – does not apply to Adam optimizer, which is adaptive
- some of the activation functions can be parametrized too
- more or better quality training data
- data preprocessing (removing silent and/or unvoiced frames)
- more features
- select only the important features so the network converges more quickly
- different (smaller) frame size during the feature extraction
- more iterations of the learning process
- feed more frames into the recurrent network

### 5.6.2 Hyperparameters

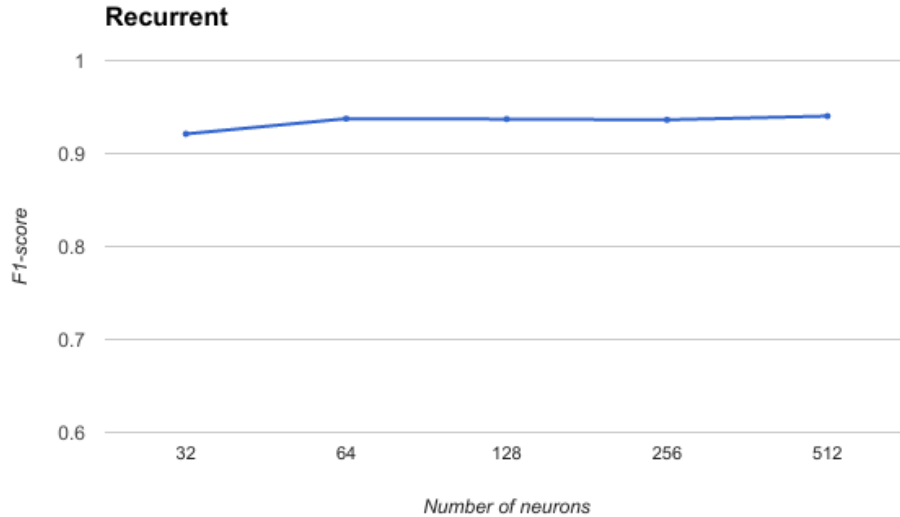
It is obvious from the previous list that there are many ways to improve the classification performance of the networks. First we look into tweaking the network architecture and enhancing its performance on unaltered training data.

## Hidden layers

The most obvious and important hyperparameters to change are depth and width of the model, i.e. number of hidden layers and number of neurons in each layer. The effect of hidden layer width can be seen in Figure 5.3. The results are measured on a network with a single layer and they show us, that there is not much, if anything, to gain by increasing the number of neurons over 128. One of the heuristics used to find out the ideal number of neurons in a hidden layer in feedforward networks says that the number should be roughly the average of input and output vector dimension, in our case  $\frac{179+6}{2} = 92.5$ , which coincides with our findings. Based on these results, if not stated otherwise, both networks will have 128 neurons per layer in the following experiments, with three layers in case of feedforward network and a single layer in case of the recurrent one.



(a) Feedforward network

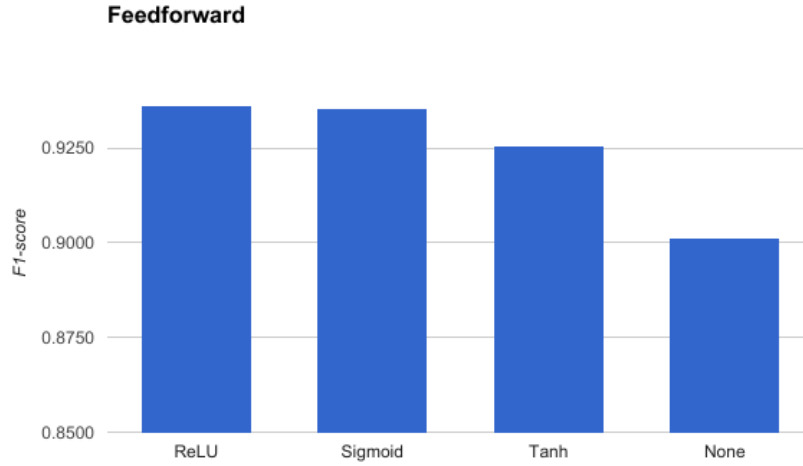


(b) Recurrent network

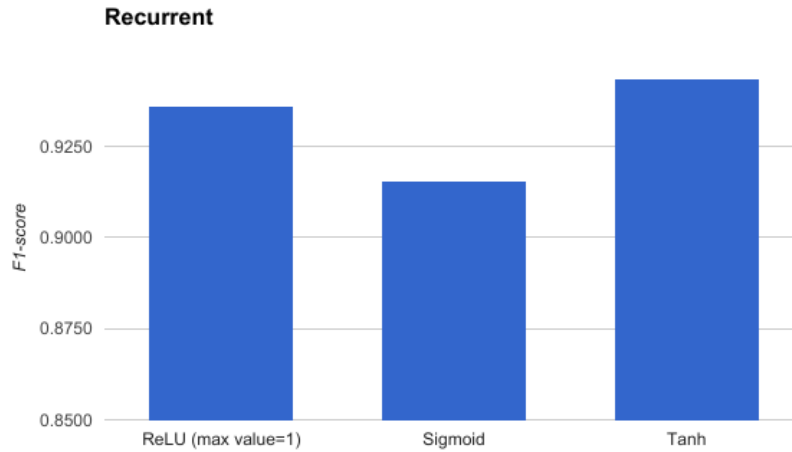
Figure 5.3: Effect of hidden layer size on classification performance

## Different activations

In the next step towards the improved results, different activation functions were evaluated. As apparent from Figure 5.4, ReLU(Rectifier Linear Unit) and tanh yield the best results for feedforward and recurrent networks respectively, so we will use these functions in further experiments. ReLU function is not bounded, which led to NaN values during the computations in recurrent networks, so it was necessary to set the maximum value to which all larger results were clipped.



(a) Feedforward network



(b) Recurrent network

Figure 5.4: F1-score while using different activation functions

## Regularization

Training f-score for feedforward network surpasses 0.99 after only a small number of epochs, while validation score stays at roughly 0.93, which provides a strong argument to use some form of regularization. GRU network yields similar values for both training and testing



phase and overfitting is not so evident here. Techniques used here are described in section 2.4.4.

**Dropout** Effects of applying dropout to the network can be seen in table 5.2 and figure 5.5.

Table 5.2: Effects of applying dropout to the networks

Dropout	Feedforward	GRU
0	0.9299	0.9357
0.1	0.9308	<b>0.9413</b>
0.2	0.9355	0.9202
0.3	0.9399	0.8247
0.4	0.9409	0.7853
0.5	<b>0.9440</b>	0.7648
0.6	0.9413	0.6457
0.7	0.9414	0.3754
0.8	0.9193	0.2315
0.9	0.6083	NaN

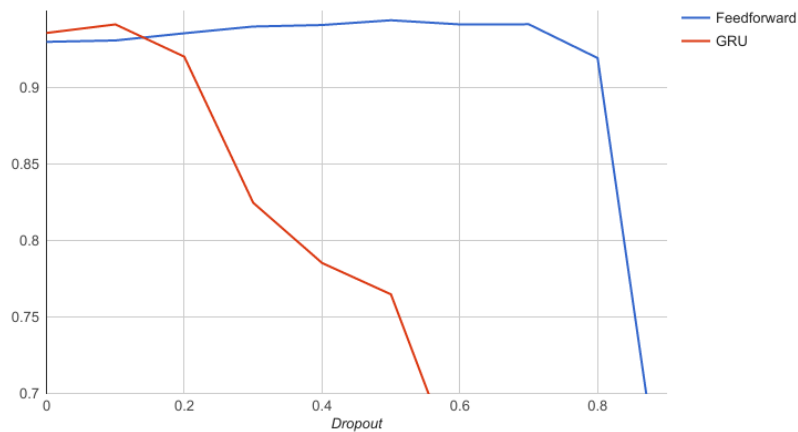


Figure 5.5: Effects of dropout on performance

### 5.6.3 Data manipulation

#### Frame size

In the baseline experiment, feature extraction was carried out on frames consisting of 1024 samples. At 8 kHz sampling rate, these frames were 128 ms long. In speech processing, the usual frame size is about 10 - 40 ms and features we used are often expected to work with these quasi-stationary frames. Shortening our frames should provide more meaningful outputs from the feature extraction and in result increases the classification score. Indeed, especially for the recurrent network, decreasing the frame size to 256 samples (32 ms) has proven to improve the accuracy notably, see Figure 5.6

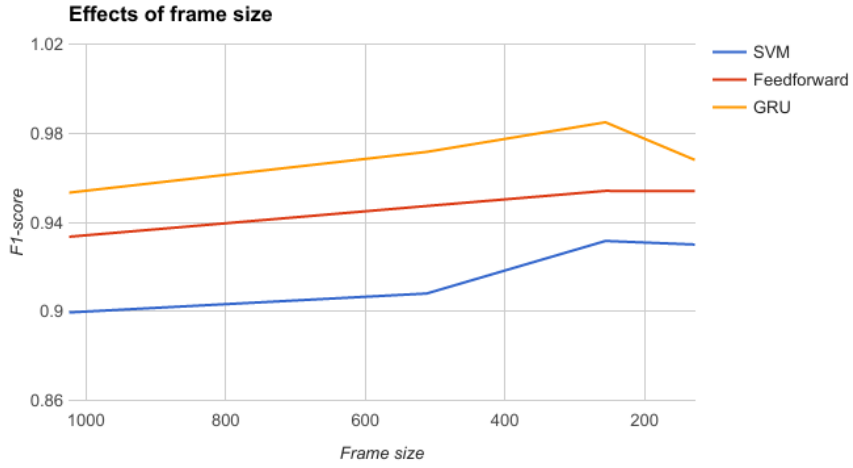


Figure 5.6: Effects of frame size

### Increasing the amount of training data

As visible from Figure 5.7 and Table 5.3, adding more training data increases the network performance notably. Doubling the number of input files yields F-score over 0.98 for the recurrent network. The results also show that the feedforward network outperforms GRU network with smaller training set, but with increased amount of samples, the recurrent network learns to generalize better.

Number of files	400	800	1200	1600
Feedforward	0.9189	0.9444	0.9538	<b>0.9609</b>
GRU	0.9145	0.9455	0.9669	<b>0.9808</b>

Table 5.3: Number of input samples

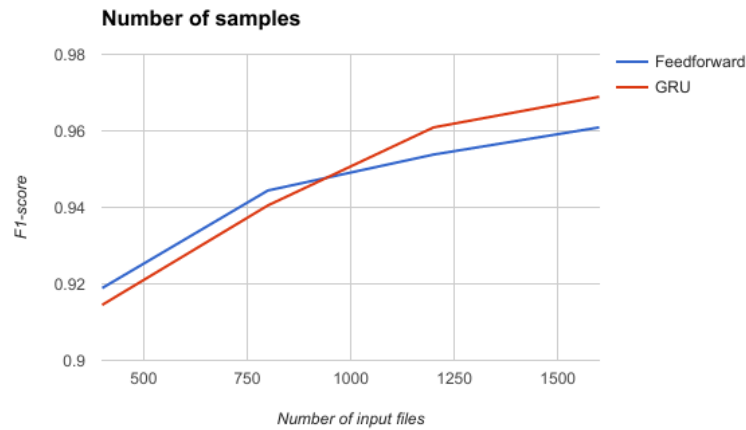


Figure 5.7: Number of input samples

#### 5.6.4 Different architecture

GRU is simplified form of LSTM, with less computational complexity, but yielding similar results on many tasks. Indeed, as seen below, there is practically no gain in performance while using LSTM instead of GRU, however, the training time of one epoch is 65% lower for GRU, due to less matrix multiplications needed. Note that this experiment was done with the optimized model from the next section, hence the higher F1 score.

Network	Epoch duration	F1 score
GRU	193 s	0.9705
LSTM	320 s	0.9711

Table 5.4: Comparison of recurrent architectures

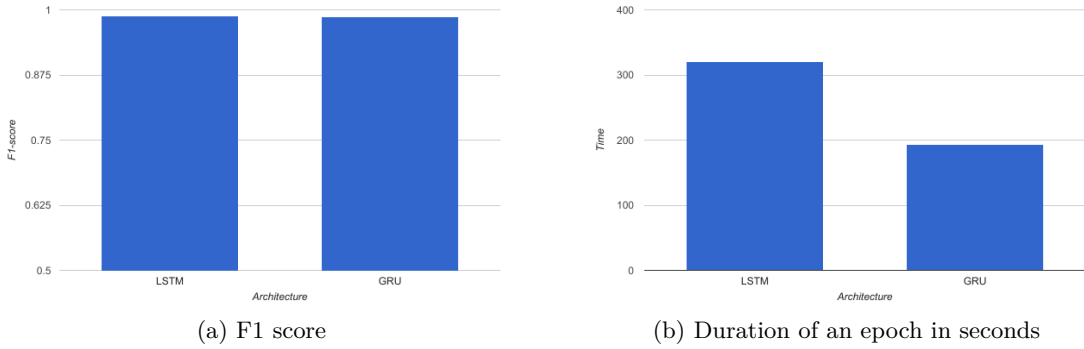


Figure 5.8: LSTM vs. GRU in terms of performance and a training epoch duration

#### 5.6.5 Optimal networks

Utilizing the observations from previous experiment, I trained the networks with the following specifications and validated it on a validation set of files that were not used in any way up to now. The results are presented in Table 5.6.

Table 5.5: Optimal network specifications

Parameter	Feedforward	GRU
Hidden layers	3	1
Layer size	128	128
Activation	ReLU	tanh
Dropout	0.5	0.1
L2 regularization	0	0
L1 Regularization	0	0
Frame size	256	256
Number of frames	600	600
Training samples	1400	1400
Validation samples	200	200

Network	F1-score	Accuracy
Feedforward	0.9645	0.9660
GRU	<b>0.9981</b>	<b>0.9985</b>

(a) Validation set

Network	F1-score	Accuracy
Feedforward	0.9089	0.9095
GRU	<b>0.9793</b>	<b>0.9805</b>

(b) Test set

Table 5.6: Results of the best network on validation and test sets

## 5.7 Further experiments

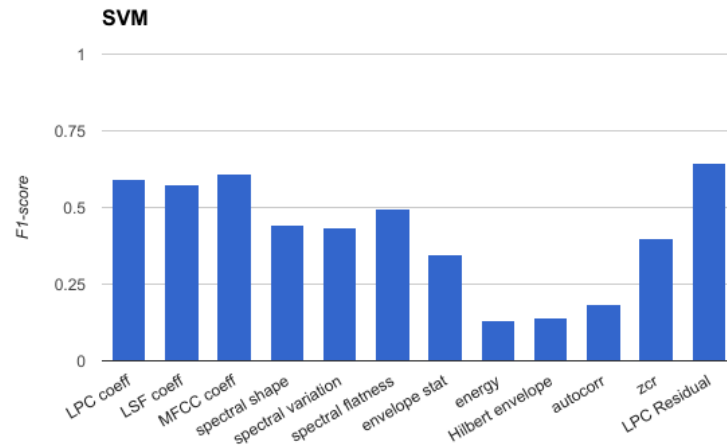
In this part, we will build upon the results from the previous part a delve into more advanced experiments using the optimal models disclosed earlier.

### 5.7.1 Features

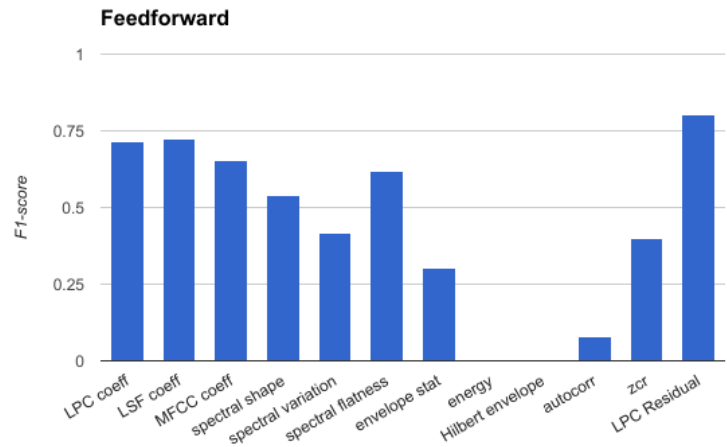
The first task will be to determine which features are the most relevant for classification. Two very simple approaches were applied; first, the classifier start with all of the features and one more was turned off for each consecutive experiment, until there was a single feature left. The second approach was to run classifier with only one feature available. As evident from table 5.7 and figure 5.9, the most significant features were LPC residual signal energy, spectral variation and spectral flatness for SVM and feedforward network and LPC coefficients for GRU network. Other important features were MFCC, Spectral shape and envelope statistics.

Feature	SVM	Feedforward	GRU
LPC coefficients	0.5933	0.7145	<b>0.9264</b>
LSF coefficients	0.5752	0.7226	0.7253
MFCC coefficients	0.6083	0.6541	0.8610
Spectral shape	0.4435	0.5406	0.9176
Spectral variation	0.4333	0.4150	0.5247
Spectral flatness	0.4950	0.6180	0.7111
Envelope statistics	0.3450	0.3013	0.8851
Energy	0.1310	-	-
Hilbert envelope	0.1410	-	-
Autocorrelation coefficients	0.1843	0.0793	0.1887
Zero Crossing Rate	0.3974	0.3991	0.1887
LPC Residual	<b>0.6450</b>	<b>0.8040</b>	0.8908

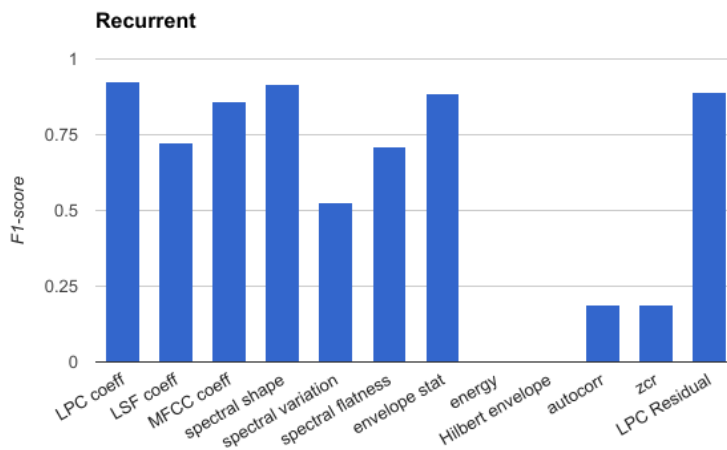
Table 5.7: Comparison of features importance



(a) SVM



(b) Feedforward network



(c) Recurrent network

Figure 5.9: Effects of particular features

### 5.7.2 „Production“ model

Based on the observations from previous sections, I simplified the best performing model to speed up the training, while keeping good classification performance. Only LPC based features were used – LPC coefficients and LPC residual signal power, variation and flatness. Other parameters were the same as in the optimal model, see Table 5.5. Table 5.8 shows that with the recurrent network it is possible to achieve F1-score and accuracy around 0.9 using only LPC based features.

Network	F1-score	Accuracy
SVM	0.7765	0.7783
Feedforward	0.8848	0.8817
GRU	0.9060	0.9017

(a) Validation set

Network	F1-score	Accuracy
Feedforward	0.7405	0.7414
GRU	<b>0.8952</b>	<b>0.8950</b>

(b) Test set

Table 5.8: Results of the „production“ network on validation and test sets

### 5.7.3 Shorter window

I also tried to find out how short can a window of samples for the recurrent network be and still be classified correctly. The network was trained using 900 frames by 256 samples, but for the tests the number of frames was gradually lowered. Also, silent frames were removed from test files by the `remove_silence.sh` script, since number of files started with a period of silence, which could have affected the results. Table 5.9 shows that the network needs at least about 10 seconds of input signal to perform feasibly.

Frames	F1-score	Duration
400	0.9428	12.8s
300	0.8942	9.6s
200	0.7105	6.4 s
100	0.6116	3.2 s

Table 5.9: Shortening the frame window for the recurrent network

## 5.8 More subsequent encodings

In this experiment I encoded the files subsequently using more codecs. I tried a number of possible combinations to study the classifier behavior. The networks were trained on the six classes as in the other experiments, the transcoded files were added only during the evaluation. The results are shown in Table 5.10. In some cases, the network classified roughly the same number of samples into both of the categories. In other cases, like when recoding GSM-EFR with G729, only the second codec was detected. Interestingly, when I trained the network with a separate class for GSM-EFR\_G729 combination, most of the samples were classified correctly, see Table 5.11.

<b>Codecs</b>	MP3	G.723.1	G729	GSM-EFR	Speex	PCM mulaw
GSM-EFR_G729	0	0	785	11	3	1
G729_GSM-EFR	0	0	332	468	0	0
G729_Speex	0	37	317	6	440	0
Speex_G729	0	20	752	14	14	0
G723.1_GSM-EFR	0	0	15	785	0	0
GSM-EFR_G723.1	0	89	173	534	1	3

Table 5.10: Classification of samples encoded by multiple codecs

<b>Codecs</b>	mp3	g.723.1	g729	gsm-efr	Speex	pcm	gsm-efr_g729
gsm-efr_g729	0	0	17	1	0	0	782

Table 5.11: Classification of samples encoded by multiple codecs when specific transcoding class is added to the training set.

## Chapter 6

# Conclusions

### 6.1 Summary

The goal of this thesis was to create a codec classification tool based on the current state of the art in the field. I managed to create a classification tool performing comparably with other papers on this topic. The result is an environment composed by number of scripts, which allowed me to experiment with different features and neural network architectures to find an optimal classifier. I succeeded to create two different approaches, recurrent and feedforward, and then increase the performance above the baseline primarily by adjusting the frame size of the feature extraction and tuning the networks hyperparameters. The final results of the best performing network seem very promising, yielding an F1-score over 0.98 on a test set.

From the perspective of the features, LPC coefficients and residual signal energy, spectral variation and spectral flatness has proven the most useful, followed by spectral shape statistics and MFC coefficients. I also found that both approaches of feature extraction are feasible – computing statistics from all the frames for a feedforward network and SVM, as well as feeding the frames one by one into a recurrent network.

Aside from the main task, I inspected behavior of the classifier in case when the sample is encoded subsequently by more encoders, and found out that it is possible to classify such sample correctly by creating a special class for specific combination of the codecs.

### 6.2 Future work

#### 6.2.1 Easily achievable goals

Some of the improvements mentioned in the experimental chapter were not implemented, for example different optimizers. However, there is probably not much to gain by implementing the rest of them, the networks perform almost perfectly on the validation set and generalize well on the test set, when enough training data is available. It would be more interesting to make the classification harder for the networks and examine its performance on more general signals.

Therefore, the first short term task is to train the models on more codecs and then evaluate them on some real signals. It is also possible to investigate their behavior when the sample is deliberately distorted, for example by adding noise. These improvements are well possible using the current scripts, the only component to change are the data.



### **6.2.2 Long term goals**

Possible long term applications may be in the field of law enforcement. In some countries, codec detection is allegedly one of the steps to verify the audio recording so that it can serve as a part of judicial proceeding. Sadly, information on whether similar systems are used in practice, are very sparse.

# Bibliography

- [1] Recommendation G.711. ITU. 1988.
- [2] ISO/IEC 13818-3. ISO/IEC. 1998.
- [3] REN/SMG-110660Q8. ETSI. 2000.
- [4] Recommendation G.723.1. ITU. 2006.
- [5] Recommendation G.729. ITU. 2012.
- [6] *Applied Data Mining and Statistical Learning*. PennState Eberly College of Science. 2017.  
Retrieved from: <https://onlinecourses.science.psu.edu/stat857/>
- [7] Bengio, Y.; Simard, P.; Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*. vol. 5, no. 2. 1994: pp. 157–166.
- [8] Besacier, L.; Bergamini, C.; Vaufreydaz, D.; et al.: The effect of speech and audio compression on speech recognition performance. In *Multimedia Signal Processing, 2001 IEEE Fourth Workshop on*. IEEE. 2001. pp. 301–306.
- [9] Chollet, F.: Keras. <https://github.com/fchollet/keras>. 2015.
- [10] Chung, J.; Gülçehre, Ç.; Cho, K.; et al.: Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR*. vol. abs/1412.3555. 2014.  
Retrieved from: <http://arxiv.org/abs/1412.3555>
- [11] Chung, J.; Gülçehre, C.; Cho, K.; et al.: Gated Feedback Recurrent Neural Networks. 2015: pp. 2067–2075.
- [12] Cybenko, G.: Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*. vol. 2, no. 4. 1989: pp. 303–314.
- [13] Dave, N.: Feature extraction methods LPC, PLP and MFCC in speech recognition. *International journal for advance research in engineering and technology*. vol. 1, no. 6. 2013: pp. 1–4.
- [14] Goodfellow, I.; Bengio, Y.; Courville, A.: *Deep Learning*. MIT Press. 2016.  
<http://www.deeplearningbook.org>.
- [15] Hahnloser, R. H.; Sarpeshkar, R.; Mahowald, M. A.; et al.: Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*. vol. 405, no. 6789. 2000: pp. 947–951.

- [16] Hicsonmez, S.; Sencar, H. T.; Avcibas, I.: Audio codec identification from coded and transcoded audios. *Digital Signal Processing*. vol. 23, no. 5. 2013: pp. 1720–1730.
- [17] Hochreiter, S.; Schmidhuber, J.: Long short-term memory. *Neural computation*. vol. 9, no. 8. 1997: pp. 1735–1780.
- [18] Jenner, F.; Kwasinski, A.: Highly accurate non-intrusive speech forensics for codec identifications from observed decoded signals. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE. 2012. pp. 1737–1740.
- [19] Jones, E.; Oliphant, T.; Peterson, P.; et al.: SciPy: Open source scientific tools for Python. 2001–. [Online; accessed <today>]. Retrieved from: <http://www.scipy.org/>
- [20] Jozefowicz, R.; Zaremba, W.; Sutskever, I.: An empirical exploration of recurrent network architectures. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*. 2015. pp. 2342–2350.
- [21] Kingma, D. P.; Ba, J.: Adam: A Method for Stochastic Optimization. *CoRR*. vol. abs/1412.6980. 2014. Retrieved from: <http://arxiv.org/abs/1412.6980>
- [22] Krizhevsky, A.; Sutskever, I.; Hinton, G. E.: Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 2012. pp. 1097–1105.
- [23] Mathieu, B.; Essid, S.; Fillon, T.; et al.: YAAFE, an Easy to Use and Efficient Audio Feature Extraction Software. In *Proceedings of the 11th International Society for Music Information Retrieval Conference*. Utrecht, The Netherlands. August 9-13 2010. pp. 441–446. <http://ismir2010.ismir.net/proceedings/ismir2010-75.pdf>.
- [24] Nielsen, M.: *Neural Networks and Deep Learning*. 2015. <http://neuralnetworksanddeeplearning.com>.
- [25] Olah, C.: Understanding LSTM Networks. <http://colah.github.io/posts/2015-08-Understanding-LSTMs/> (visited: 31.5.2017). 2015.
- [26] Scholz, K.; Leutelt, L.; Heute, U.: Speech-codec detection by spectral harmonic-plus-noise decomposition. In *Signals, Systems and Computers, 2004. Conference Record of the Thirty-Eighth Asilomar Conference on*, vol. 2. IEEE. 2004. pp. 2295–2299.
- [27] Sharma, D.; Naylor, P. A.; Gaubitch, N. D.; et al.: Non intrusive codec identification algorithm. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*. IEEE. 2012. ISBN 9781467300469. pp. 4477–4480.
- [28] Silva, D. F.; de Souza, V. M. A.; Batista, G. E. A. P. A.: A comparative study between MFCC and LSF coefficients in automatic recognition of isolated digits pronounced in Portuguese and English-doi: 10.4025/actascitechnol. v35i4. 19825. *Acta Scientiarum. Technology*. vol. 35, no. 4. 2013: pp. 621–628.

- [29] Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; et al.: Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*. vol. 15, no. 1. 2014: pp. 1929–1958.
- [30] Sun, L.; Mkwawa, I.-H.; Jammeh, E.; et al.: *Guide to Voice and Video over IP: For Fixed and Mobile Networks*. Springer Publishing Company, Incorporated. 2015. ISBN 1447158431, 9781447158431.
- [31] Valin, J.-M.: Speex: a free codec for free speech. *arXiv preprint arXiv:1602.08668*. 2016.
- [32] Zaremba, W.; Sutskever, I.; Vinyals, O.: Recurrent Neural Network Regularization. *CoRR*. vol. abs/1409.2329. 2014.  
Retrieved from: <http://arxiv.org/abs/1409.2329>

# Appendix A

## Cookbook

### A.1 Software and modules

Necessary software is summarized in this section.

#### A.1.1 Python

Python Version: 2.7.13

##### External modules

Name: Keras

Version: 1.1.1

Used to create neural networks.

Name: tensorflow

Version: 0.10.0rc0

Keras runs on top of either Tensorflow or Theano libraries, Tensorflow was used in this thesis.

Name: NumPy

Version: 1.12.0

Necessary for the matrix operations.

Name: SciPy

Version: 0.18.1

Data preprocessing, SVM.

Name: Yaafe

Version: 0.64

Feature extraction.

Name: PySoundFile

Version: 0.8.1

Loading sound files into NumPy arrays.

### A.1.2 Other tools

**The Edinburgh Speech Tools Library** [http://www.cstr.ed.ac.uk/projects/speech\\_tools/](http://www.cstr.ed.ac.uk/projects/speech_tools/)

**FFmpeg** Version:2.8.11 Encoding the files, removing silence.

**GSM-EFR encoder from ETSI**

**G729 encoder**

### A.1.3 Scripts

**get\_corpora.sh** Downloads the corpora and creates the necessary directory structure.

**recode.sh** Encodes the files obtained through *get\_corpora.sh* with ffmpeg and other encoders, creates the training/test sets.

**lpc\_extract.sh** Extracts an LPC residual signal from specified files.

**do\_lpcs.sh** Calls *lpc\_extract.sh* on data prepared by *recode.sh*

**remove\_silence.sh** Removes frames with volume level under threshold from specified files.

**resample.sh** Resamples 16 kHz files from the corpora to 8 kHz.

**do\_everything.sh** Should do everything, assuming that the paths to Edinburgh speech tools and GSM-EFR and G729 encoders are set correctly on its first three lines – download the corpora, resample, encode and extract LPC residual signal. After running this script, the environment is prepared for running the classifier.

**classify.py** Builds the models and trains them. Possible parameters:

- test* Do not train the model, only classify the input files. Must be used together with *-ffweights* and *-gruweights*
- files=* Number of files to process from each of the input directories
- outffweights=* Path to a file to save the weights from the feedforward network
- outgruweights=* Path to a file to save the weights from the recurrent network
- ffepochs=* Number of epochs to train the feedforward network
- gruepochs=* Number of epochs to train the recurrent network
- l1=* L1 regularization coefficient
- l2=* L2 regularization coefficient
- ffweights=* File with weights for feedforward network

*-gruweights*= File with weights for GRU network

*-help* Print help

*-save\_features*= Prefix of files to save the NumPy arrays of features for both networks to.

*-ffdropout*= Feedforward dropout.

*-input*= Prefix of the two files to load the NumPy arrays of features from.

*-grudropout*= GRU dropout.

*-fflayersize*= Feedforward layer size (number of neurons)

*-grulayersize*= GRU layer size

*-frames*= Number of frames to use as an input for the recurrent network.

*-frame\_size*= Number of samples in each frame

*-optimizer*= Optimizer to use for the networks.

#### A.1.4 Step by step

1. Preparing the files should be fairly straightforward – only things that must be set are paths to Edinburgh Speech Tools, G729 codec and GSM-EFR codec. These paths can be set in the first lines of `do_everything.sh`. In case the script is unsuccessful, it is possible to run the steps separately:
  - (a) `get_corpora.sh`
  - (b) `resample.sh`
  - (c) `recode.sh`
  - (d) `do_lpcs.sh`
2. Now it is possible to run `classify.py`, which is documented by the Readme file and by source code comments.